Software Engineering Institute

Case Study: Model-Based Analysis of the Mission Data System Reference Architecture

Peter H. Feiler (Software Engineering Institute)
David Gluch (Software Engineering Institute)
Kurt Woodham (L-3 Communications-Titan Group)

May 2010

CarnegieMellon

Table of Contents

List of Figures

List of Tables

Acknowledgments

Executive Summary

The aerospace industry is experiencing exponential growth in the size and complexity of onboard software. It is also seeing a significant increase in errors and rework of that software. All of those factors contribute to greater cost; the current development process is reaching the limit of affordability for building safe and reliable aircraft and spacecraft. The size of software in aircraft with respect to source lines of code (SLOC) has doubled every four years since the mid-1990s; the 27 million SLOC projected for 2010-2020 is estimated to cost more than $10 billion. Studies into the role of software in spacecraft accidents and the increasing complexity of flight software indicate the need for improvement in requirements elicitation and architecture, in particular for validation early and throughout the life cycle through modeling and analysis that complement testing.

In order to improve predictability, the system and software engineering communities are practicing model-based engineering, where models of different aspects of a system are developed and analyzed. However, industrial experience has shown that such models, developed independently over the life cycle, result in multiple versions of the "truth" (i.e., they are not consistent with each other and the evolving architecture). The SAE Architecture Analysis and Design Language (AADL) standard addresses this issue of multiple truths due to inconsistency between analytical models by providing an architecture modeling notation with well-defined semantics that can accommodate multiple analysis dimensions through annotations and allow for auto-generation of these analytical models from a single source.

The Carnegie Mellon® Software Engineering Institute, L-3 Communications - EITS, and the Jet Propulsion Laboratory (JPL) have collaborated in a use of model-based engineering for the National Aeronautics and Space Administration (NASA) Software Assurance Research Program (SARP) project named "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." The work involved applying the AADL to the Mission Data System (MDS) architecture. The SAE AADL industry standard for modeling and analysis of embedded software system architectures was chosen because of its ability to support analysis of non-functional properties, such as robustness, safety, performance, and security. The MDS was chosen because it takes an architecture-centric view by defining a multi-layered reference architecture for autonomous systems, whose dynamics are managed by feedback loops, and promotes state analysis through goal-oriented modeling to address uncertainty and faults. By combining the two technologies, we can take into account the impact of the embedded software's runtime architecture on these non-functional properties in the validation of systems.

The result of that project shows that the AADL can

- effectively model MDS top-level constructs (e.g., hardware adapters, separation of estimation and control, the layering of planning and control)
- effectively represent the MDS reference architecture and support an instantiation of this architecture for a sample system

---

- address key MDS architectural themes (e.g., state-based closed loop control, separation of estimation from control and data management from data transport, ground-to-flight migration)
- provide a foundation for the analysis of critical MDS performance elements and system assurance concerns (e.g., latency, task scheduling, integral fault protection)

Abstract

This report documents the results of applying the Architecture Analysis and Design Language (AADL) to the Mission Data System (MDS) architecture. The work described in this case study is part of the National Aeronautics and Space Administration (NASA) Software Assurance Research Program (SARP) research project "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." The report includes discussion of modeling and analyzing the MDS reference architecture and its instantiation for specific platforms. In particular, it focuses on modeling aspects of state-based system behavior in MDS for quantitative analysis. Three different types of state-based system models are considered: closed loop control, goal-oriented mission plan execution, and fault tolerance through mission replanning. This report demonstrates modeling and analysis of the MDS reference architecture as well as instantiations of the reference architecture for a specific mission system.

1 Introduction

This document presents the results of a case study of the application of the Architecture Analysis and Design Language (AADL) to the Jet Propulsion Laboratory (JPL) Mission Data System (MDS). The work under this project is a collaboration of the Carnegie Mellon® Software Engineering Institute, L-3 Communications - EITS, and the Jet Propulsion Laboratory (JPL). This effort is part of the National Aeronautics and Space Administration (NASA) Software Assurance Research Program (SARP) project "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." The project is an expansion and continuation of the work completed under the NASA IV&V Facility Funded project "The Application of SAE Architecture Analysis and Design Language (AADL) to IV&V of NASA Flight Projects."[1]

The work described in this case study is motivated by the effects of exponential growth in the size and complexity of onboard software. Studies have revealed the increased role of software in spacecraft accidents [Leveson 2004] and an increase in complexity in flight software [NASA 2009]. Furthermore, industry statistics indicate that the size of software in aircraft measured in source lines of code [SLOC] has doubled every four years since the mid-1990s and is expected to grow to 27 million SLOC for the decade of 2010-2019 at a cost of more than $10 billion. Those statistics also show significant increases in errors and rework of embedded software: 70% of the errors are introduced early, during requirements specification; however; 80% of the errors are detected and repaired later and at higher cost, during system integration, acceptance test, and operation [SAVI 2009].

Model-based software assurance is the application of model-based engineering techniques (i.e., the use of models and abstractions to perform typical engineering tasks) to the verification and validation of software. Model-based software assurance relies on analytical practices using analysis and modeling languages and supporting tools. Validation through analysis and simulation of models in the application domain is common practice (e.g., the use of Simulink for controls systems, and state analysis in MDS [Ingham 2004]). The AADL [SAE AADL 2004/2009] and its supporting tools such as the Open Source AADL Tool Environment (OSATE) [SEI 2010] have been designed to capture the architecture of embedded software systems in terms of the application software as a runtime architecture deployed on a particular computer system. This model allows V&V personnel to develop a thorough understanding of and insight into (1) critical characteristics vital to a system's correct operation and (2) the impact the runtime architecture and computer system deployment on the non-functional system properties. These characteristics include considerations such as sensor/command data latency and update rates; CPU throughput; synchronous/asynchronous task management; and data-bus packet definitions and update rates.

The models and analyses presented in this report are results of a case study effort that is applying the AADL to represent and analyze the MDS, with a focus on state-based system behavior. The report is organized as follows:

- Section 2 provides an overview of the MDS architecture and observations on MDS architectural themes as they relate to AADL technology and state-based system modeling.

- Section 3 presents AADL models of the MDS reference architecture and discusses specific approaches for organizing the AADL model and for modeling the multi-layered architecture in AADL in terms of a control system, a system under control, and a computer platform.

- Section 4 presents an instance of the MDS architecture—namely the heated camera system example extracted from the Jet Propulsion Laboratory tutorial "Software Architecture Themes in JPL's Mission Data System" [Bennett 2006]—and illustrates two examples of system engineering analysis (mass analysis and coarse-grained power analysis).

- Section 5 discusses how (1) the closed loop control layer of the MDS architecture can be modeled as a flow-oriented architecture while preserving the notion of control states and (2) this model can be utilized in performing end-to-end latency analysis on the control loop.

- Section 6 discusses an approach to model how the planning layer can be used to
  - drive the execution of the control layer based on a goal network
  - perform analysis of the workload the control layer tasks impose on the computer platform

- Section 7 discusses how management of mission goal failures, replanning, and reconfiguration of controllers can be represented in the AADL model and included in a processor resource analysis.

- Section 8 provides a summary of the key insight of this case study.

In addition, the appendix includes detailed listings of AADL textual models, and a glossary of acronyms and list of references are included at the end of this document.

2   Mission Data System Overview

The JPL initiated the MDS project in April 1998 with these principal objectives:

1. "to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions"

2. to address "several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolvability to *in situ* exploration and other autonomous applications" [Dvorak 2000]

This section provides an overview of the key MDS architecture themes, the state-based control approach, the separation of concerns within the MDS architecture, and their relation to AADL and model-based engineering.

2.1   State-Based Behavior in MDS

The MDS approach is characterized by a set of architecture themes [Dvorak 2000]. In this section, we associate AADL-specific capabilities with some of the MDS architecture themes. These observations provide guidance in developing analysis strategies and approaches, identifying critical issues, and defining specific views and models for the MDS case study. The MDS themes and AADL-specific capabilities are summarized in Table 1.

Table 1:   MDS Architectural Themes and Associated AADL Capabilities

Theme	Description	AADL Capabilities
Take an Architectural Approach	Construct subsystems from architectural elements, not the other way around	AADL is an architecture description language for real-time systems.
Ground-to-Flight Migration	Migrate capability from ground to flight, when appropriate, to simplify operations	AADL clearly separates the application architecture from its deployment on physical and computer platforms.
State & Models are Central	System state and models form the foundation for information processing.	AADL supports modeling of systems using shared variables as well as flow-oriented modeling through ports and connections. This report maps the MDS state variable view into a flow-oriented view.
Explicit Use of Models	Express domain knowledge explicitly in models rather than implicitly in program logic	AADL is a formal language that supports rigorous modeling of systems as software and hardware components and their interactions.
Goal-Directed Operation	Operate missions via specifications of desired state rather than sequences of actions (Goals are constraints on state variables over a time interval.)	The goal network representation of a mission plan acts as a task plan that is interpreted by the goal executive. The result is a set of action requests to the control layer of the system.
Closed Loop Control	Design for real-time reaction to changes in state rather than for open-loop commands or earth-in-the-loop control	AADL supports modeling of closed loop flow-oriented architectures through data ports for deterministically communicating state, including mid-frame and phase-delayed state transfer.

Theme	Description	AADL Capabilities
Resource Management	Resource state usage is projected against models and checked against the constraints.	AADL properties enable the characterization of resource capacities and resource budgets. AADL multi-layer modeling allows explicit modeling of resource management capability.
Separate State Determination from State Control	For consistency, simplicity and clarity, separate state determination logic from control logic.	AADL packages and components allow users to organize and compartmentalize the model (i.e., separate state determination from state control).
Integral Fault Protection	Fault protection must be an integral part of the design, not an add-on.	AADL specifies fault handling behavior of threads, supports fault management patterns, and the Error Model Annex models fault behavior to support dependability and safety criticality analysis.
Acknowledge State Uncertainty	State determination must be honest about the evidence that state values are rarely known with certainty.	Data component types and AADL properties can be used to characterize the data represented in state variables including uncertainty characteristics.
Separate Data Management from Data Transport	Data management duties and structures should be separated from those of data transport.	AADL supports modeling in abstractions that separate data management from data transport and data history logging.
Join Navigation with Attitude Control	Navigation and attitude control must build from a common mathematical base.	AADL components are abstractions of system components with characteristics mapped into properties.
Instrument the Software	Instrument the software to gain visibility into its operation, not just during testing but also during operation.	AADL provides properties and patterns to model instrumentation of software.
Upward Compatibility	Design interfaces to accommodate foreseeable advances in technology.	AADL semantics allows the partial description of component interfaces that can be specialized within implementations or extensions. In addition, properties of these interfaces can be used to explicitly capture compatibility requirements.

Several aspects of the MDS architecture can be characterized as different forms of state-based systems, each with its own semantics.

- **State & Models are Central:** States and models allow engineers to focus on the goals of a mission instead of the individual actions to be taken to achieve a goal. In control systems, observed state and desired state drive the system under control. The stability of control algorithms is affected by variation of the latency and age of data being processed.

- **Closed Loop Control:** MDS has a closed-loop control layer. It operates on sensor measurements and observations to drive a controlled system towards desired goals through actuator output. This flow of state information in a control system can be directly modeled in AADL as flow through data ports and connections instead being implicit in the access patterns to state variables. This flow-oriented view allows us to investigate how the runtime system of software impacts the latency assumed by a control engineer.

- **Goal-Directed Operation:** MDS uses a goal-oriented approach to operate missions by specifying desired state. Interpretation of a goal network by a plan execution engine together with a monitoring component results in issuing service requests to the components in the

closed-loop control layer. In this context, we will illustrate analysis of the workload on the processor generated by the execution of the goal network against available resource capacities.

- **Resource Management:** Resource management takes on these forms in MDS:
  - During mission planning, analysis of the goal network takes into consideration constraints for maximizing utilization of limited consumable resources such as power.
  - Execution of the goal network results in different workloads generated by the control layer, as different sets of control activities are requested at different points in time.
  - Observed failure to reach a goal results in replanning and plan merging, activities that add to the workload at any given point in time.

  AADL processors, memory, bus, and device concepts represent computer platform and physical resources. The use of properties allows users of AADL to characterize available resource capacities by hardware and required resource budgets by software. At any given time, the processor speed may be adjusted to reduce power consumption while still meeting timing requirements when processing a given workload.

- **Integral Fault Protection:** MDS integrates fault protection into the goal network planning process. Exceptional conditions in the environment and faults in the system under control are addressed by alternate goals and by replanning when goals become unreachable. Faults in the computer hardware and application software may require a secondary layer of fault management in the form of reconfiguration of the application software deployment and computer hardware. AADL includes fault handling mechanisms as part of its execution semantics including recovery entry points for threads to take recovery action, error event ports to communicate faults to a health monitor, and modes to represent various fault tolerant and fault recovery configurations to protect against and respond to faults (i.e., it allows users to model fault monitoring and management as supported by the system). AADL also has an Error Model Annex extension that permits users to abstractly characterize fault behavior and fault propagation in support of fault impact and isolation analysis as well as reliability and fault tree analysis.

- **Separate Data Management from Data Transport:** AADL separates data management from data transport by
  - representing the intended interaction of components through information exchange via ports and connections as well as shared access to persistent data components
  - binding the components and their interactions to processors, memory, and buses

  Buses represent the physical communication medium as well as the protocols used for transport. Connections can have properties that express the transport requirements, such as guaranteed delivery or ordered delivery.

2.2 State-Based Control

State-based control for the MDS is a goal-driven control approach where the control system and underlying software implementations are planned by a user through goals, rather than actions, and then translated into commands. Goals are constraints on the value of a state variable over a time

interval. State variables represent quantified observations that describe the condition of the system under control (e.g., temperature, velocity, or position).

The themes shown in Table 1 coupled with the state-based goal-driven approach define the MDS control system architecture that is shown in Figure 1. The software and hardware implementation of this architecture is driven by the State Analysis systems engineering methodology. State Analysis is a process for capturing system and software requirements in the form of explicit models [Ingham 2004].
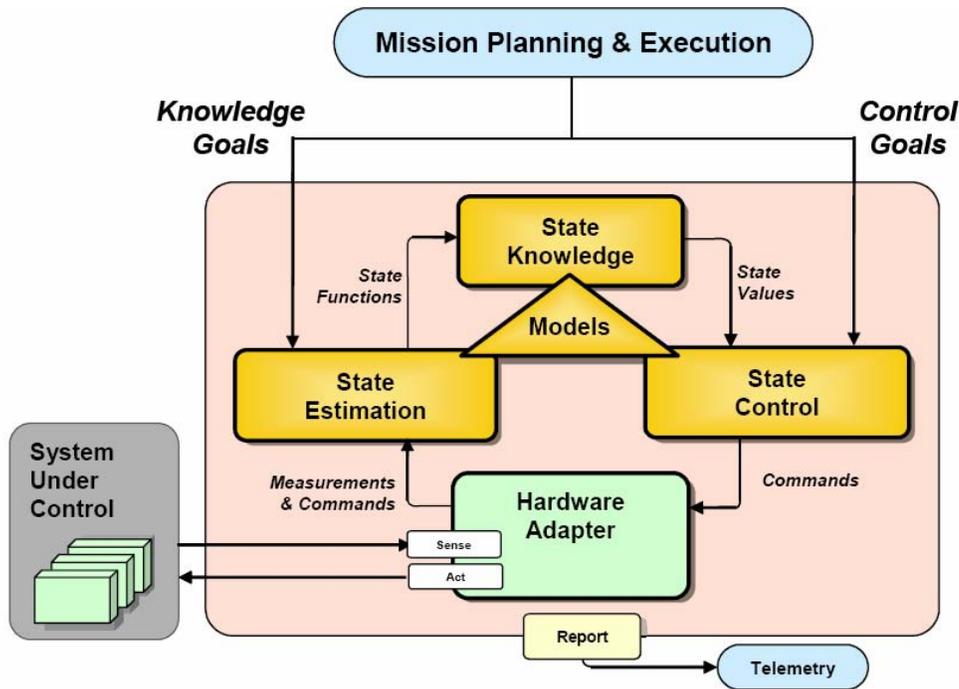
Figure 1: MDS Control System Architecture [Bennett 2006]

2.3 Separation of Concerns

The separation of concerns and isolation of interfaces in the MDS architecture is shown in Figure 2. In this partitioning, the control system is distinct from the system that is under control and the control system's elaboration, projection, and scheduling are separated from the execution of control. The execution of the control is separated into goal execution and monitoring, estimation, and active control.

2.4 MDS Layered Architecture

Figure 3 presents the layered structure of the MDS architecture extending from the system under control up through four layers: scheduling, goal elaboration & re-elaboration, controllers, and estimators. In addition to showing the separation of the *goal elaboration & re-elaboration* components of the planning layer, the *goal executive* & *monitor* components are explicitly identified within the execution layer.
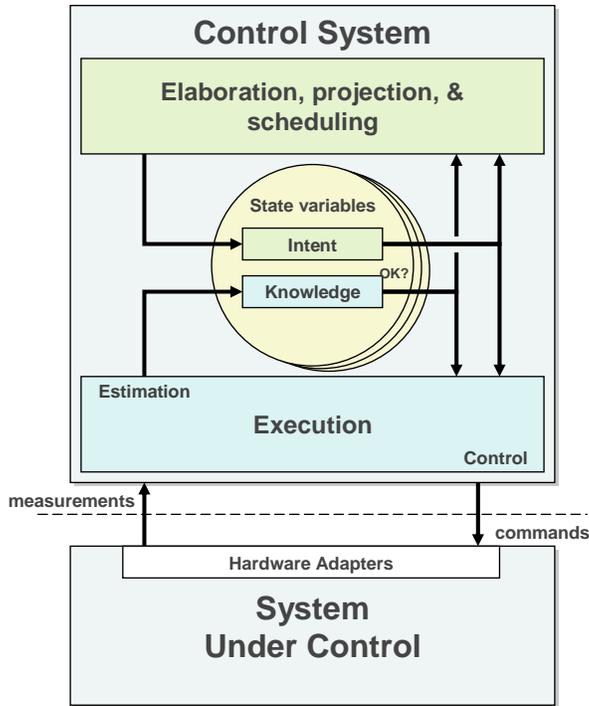
Figure 2:   MDS Architectural Separation of Concerns [Bennett 2006]
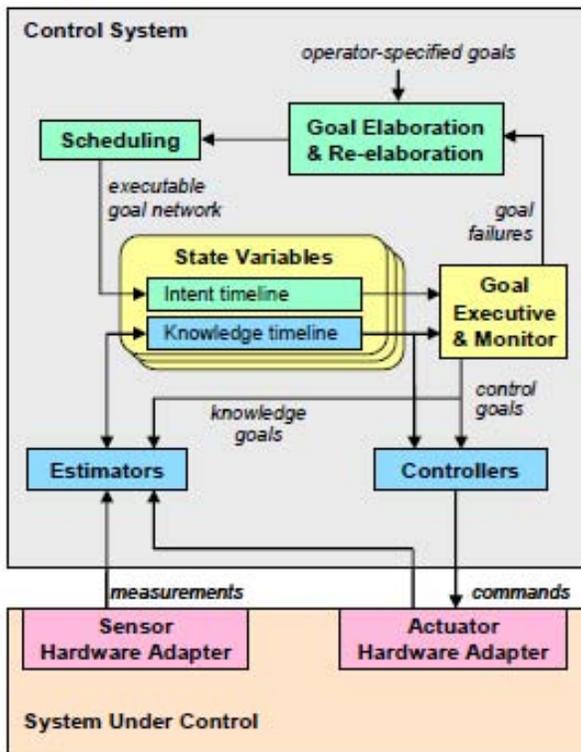
Figure 3:   MDS Layered Architecture [Bennett 2008]

3 AADL Model of the MDS Reference Architecture

In this section, we present the core elements of the MDS architecture as an AADL model in a way that allows the model to be refined into a specific instance of an MDS. This refinement is discussed in Section 4.

3.1 Top-Level MDS Representation

The top-level architecture is shown in Figure 4 and reflects Figure 2 from page 7. The system under control is represented by the AADL system component *MDSSystemUnderControl* and the control system is shown as an AADL process called *MDSControlSystem*. For the top-level model we have followed the view of the MDS architecture presented in Figure 2, with the system under control consisting of the physical system as well as the hardware adapters that convert the sensor readings into normalized measurements and the control commands into commands in actuator-specific formats. The hardware adapters maintain the value history of the measurements and commands.
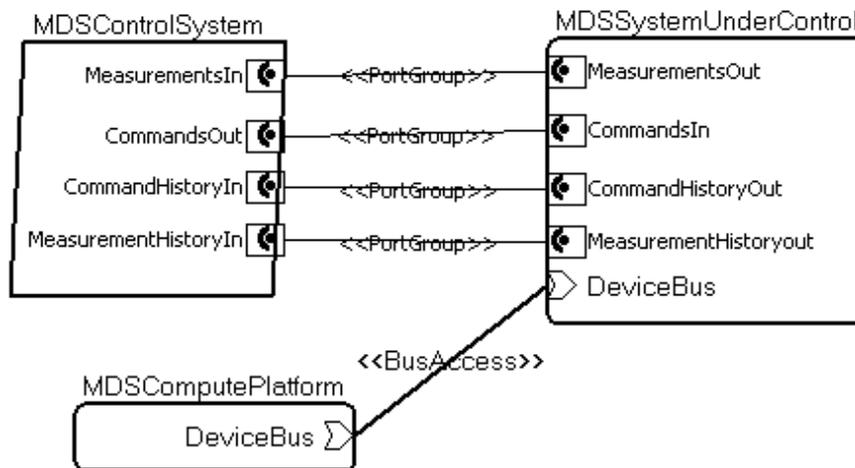
Figure 4: Top-Level MDS Architecture

These two components interact by passing measurements and control commands. Those interactions are shown as connections between port groups to indicate that there may be a collection of connections between the two components. When the MDS architecture is instantiated, the measurements port group is refined into a collection of data ports, one for each measurement. In the same way, the control commands port group is refined into a collection of data ports to represent commands sent to the actuator adapters. Similarly, the value histories of measurements and commands are made available to the MDS control system through port groups that get refined for a specific MDS instance.

Unlike the representation in Figure 2, in Figure 4 we show the computing platform for MDS explicitly as a third component. It may include the flight system computer platform and the ground system computer platform and their connectivity. The computing platform is connected to the system under control through a device bus that provides physical access to the sensors and actuators

in the system under control. The MDS software components of the MDS control system and of the system under control are mapped to the computer platform in a deployment configuration through the use of AADL binding properties.

3.2 System Under Control

The MDS system under control consists of the physical system being controlled (i.e., its sensors, actuators, and hardware adapters). As shown in Figure 5, the sensor readings are passed from the physical system to the hardware adapters for conversion into measurements. Similarly, the control commands are converted by the hardware adapters and passed to the actuators in the physical system. The hardware adapters maintain measurement and command histories and make them available.

For a specific MDS instance, the port groups shown in the reference architecture will be refined to represent specific sensor output and actuator input, measurements and commands, and histories.
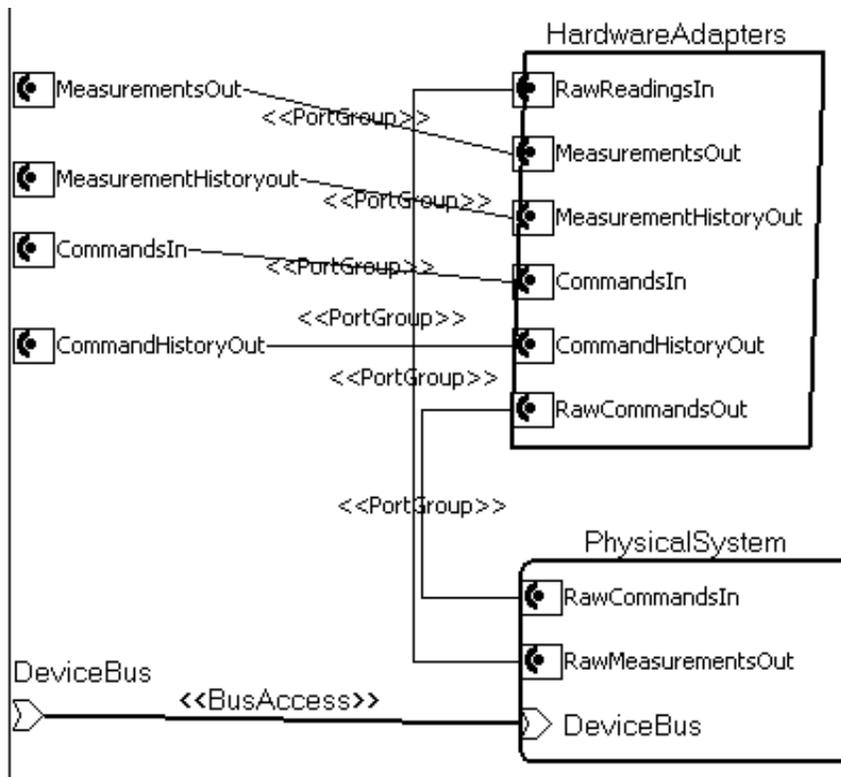
Figure 5: System Under Control in MDS Reference Architecture

When the MDS architecture is instantiated for a specific system, the physical system under control is refined. In this refinement, details of AADL devices that represent the plant being controlled are defined. The plant can be modeled at different levels of detail as appropriate. A single device may represent the complete plant. In that case, *out* data ports represent sensors whose data content are measurements, and *in* data ports represent actuators whose data content represent actuator commands. One can also choose to represent each sensor and each actuator as a separate device. Each device would include one or more ports for measurements or commands. In this

model, properties can be associated with each sensor, actuator, and plant to indicate power consumption, failure rate, and other physical characteristics.
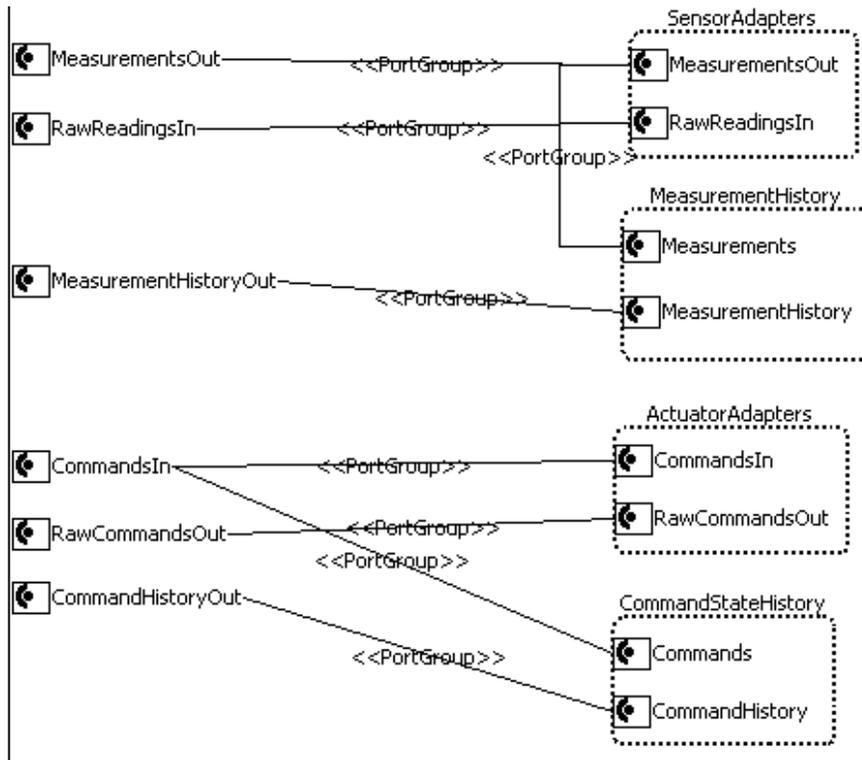
Figure 6:   Sensor and Actuator Adapters and Value History

Figure 6 shows the details of the hardware adapters in the MDS reference architecture. We have transformed the actual adapters into sensor adapters, which are responsible for converting sensor readings into measurements, and actuator adapters, which are responsible for converting control commands into actuator commands. In addition, we have included value history stores for measurement and command histories. All these components will get refined with specific adapters for a MDS instance.

3.3 The Control System

The AADL model of the MDS control system, which is shown in Figure 7, reflects the layered architecture of Figure 3 from page 7. In the MDS architecture, state variables are used as the communication channel (container) through which information such as state estimates (knowledge), goals (intent), measurements, and commands is communicated between components of the MDS architecture, as shown in Figure 2. AADL allows us to abstract such communication channels into the underlying platform through the AADL virtual bus concept and express the information flow as port connections between the components. Port connections reflect the source and the recipient of information. The fact that information such as state estimates is communicated through a connection is represented by the fact that the connection is bound to a particular virtual bus.

In this model, we decided to show the measurement stream and the measurement history through separate port groups to more precisely reflect who is using the sensor data stream and who needs

to operate on the history. For more on our modeling of data streams, value histories, and data state variables in AADL see the Section 3.4.
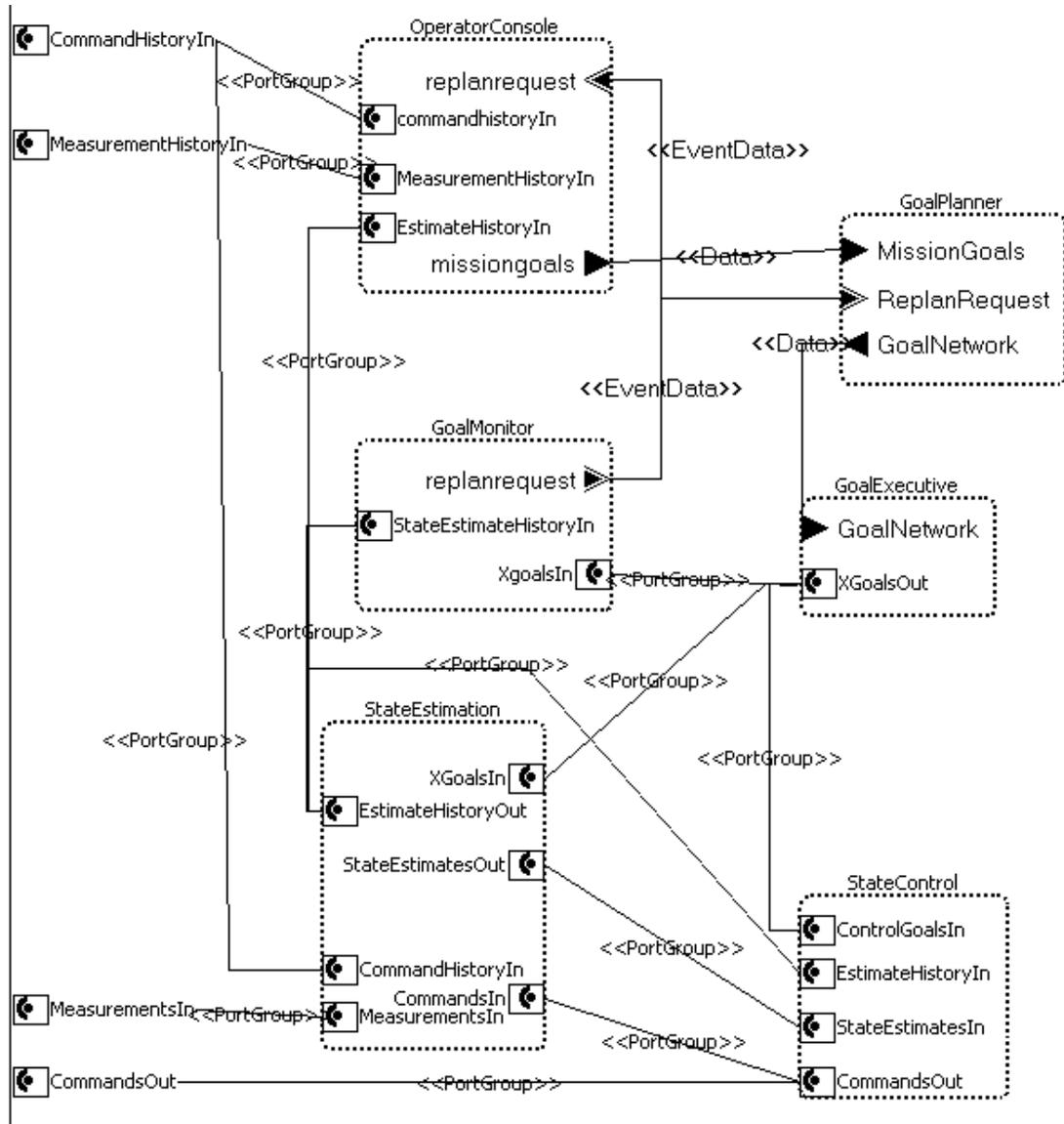
Figure 7: The MDS Control System

The bottom layer represents the state estimators and the controllers of the control layer in Figure 3. The estimators are represented by the *StateEstimation* thread group and the controllers are represented by the *StateControl* thread group. Bundling these as thread groups allows the refinement of each with a set of threads that represent individual estimators and controllers when the MDS architecture is instantiated for a specific system. The port group *StateEstimatesOut* represents the results of the estimators (i.e., the observed state of the system under control). This port group is refined with data ports, each acting as the current value of an estimation state variable. The state estimates are made available to the controllers (*StateControl*). The *StateEstimation*

thread group is also responsible for maintaining a history of the estimated states. This history is made available through a separate port group *EstimateHistoryOut*.

The second layer represents the goal executive and the goal monitor of the execution layer in Figure 3. The goal executive interprets a goal network (i.e., a mission plan) and passes *Xgoals* to the controllers. The goal monitor compares the state estimate history against the *Xgoals* to determine whether the controllers are unable to reach their goals and replanning should be initiated. The *Xgoals* are represented by a port group that is refined when the MDS architecture is instantiated for a specific system.

The top of Figure 7 represents the goal planner (i.e., the planning layer and the operator console—the presentation layer of Figure 3). The goal planner is responsible for producing a goal network and for replanning (i.e., re-elaborating the goal network, if the controllers are unable to meet their goals within the goal network constraints). The operator console provides status and allows for goal planning input.

3.4 Data State Variables, Value History, Data Control, and Telemetry

A data state variable is a key concept of MDS for representing information about the data being processed by MDS (i.e., about estimated, intended, and projected physical state). It allows for characterizing

- data truths about value histories of measurements, commands, and estimated states

- intended and projected states reflected in the goal network

A common way of modeling such meta-information in AADL is to associate AADL properties with the item in question and record information about the item. For example, the measurement unit and confidence of data may be recorded in properties. Since AADL is extensible and allows us to introduce new properties, we can define a set of properties specific to the data state variable. In some cases, this Meta information is communicated explicitly with the data and is checked by the application at runtime. In this case, the Meta information is declared to be part of the data representation, either just reflected in the increased size of the data type, or explicitly as a data subcomponent in a data component implementation declaration.

In the MDS, hardware adapters maintain a value history of measurements and control commands. Similarly, value histories of estimated state are maintained. In the package *ValueHistories* we have introduced a set of components that represent value histories and functionality of updating it and making it accessible. The value history stores for measurements and commands are kept with the hardware adapters, while those of estimated state are kept with the estimators. We use history port groups as an abstraction of how value history is made accessible. This allows us to capture that the history may be made available in the form of state functions.

The MDS has a data control component whose responsibility is to manage the value history as a data resource according to constraints provided by an engineer. Similarly, there is a data transfer component whose responsibility is to move data between deployments. The data control and data transfer components can be modeled in AADL at different levels of fidelity. When modeled abstractly, we can simply define, through AADL properties, attributes of the data ports that identify value history requirements. When the ports are bound to memory, their memory requirements are determined, not only by the data type of the state value but also by value history requirements.

Desired compression strategies can be specified via properties on a data port or on its data component type. In this case, it is assumed that the underlying runtime system represented by the computer platform supports this logging capability.

The MDS State Variables hold observations (knowledge) determined by estimators and plan goals (intent). These state variables are communicated between Ground and Flight systems via telemetry. The data transport mechanism uses State Variables and State Variable Proxies. A State Variable represents the location in the deployment where the state is being locally estimated, and a Proxy State Variable represents a remote location that intends to utilize state variable content remotely. This deployment is shown in Figure 8.
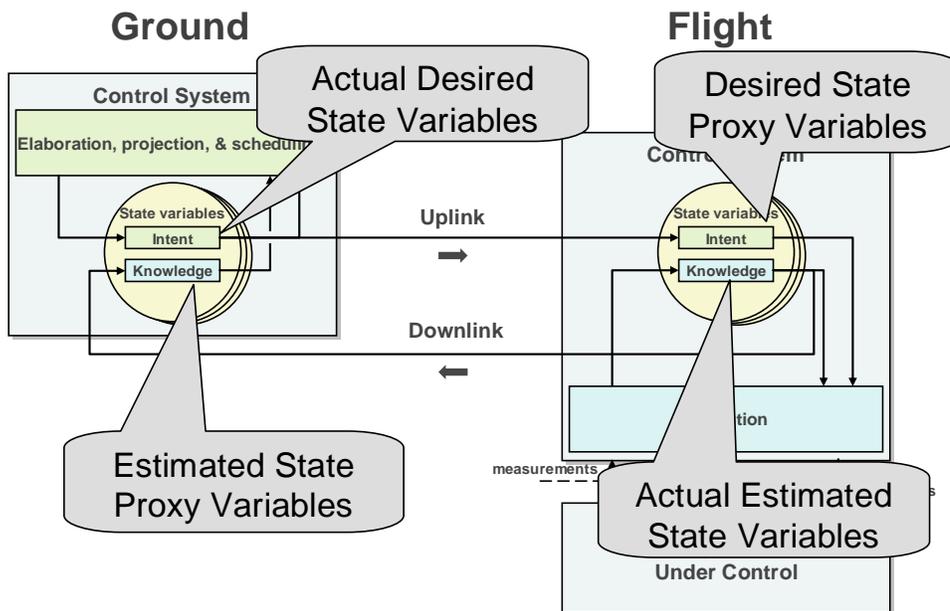
Figure 8:   Deployment of State Variables [Bennett 2006]

The deployment of these data is such that

Estimators in a deployment update their corresponding State Variables (SV). The data transport mechanism occasionally collects the value histories stored in these SVs and transports these histories to appropriate Proxy SVs in other deployments. The same data transport mechanism is used to transport measurement histories and command histories between deployments (from Basis Hardware Adapters to Proxy Hardware Adapters). Systems engineers specify what information needs to be transported between deployments, and the regularity of proxy updates [Bennett 2006].

The telemetry transport mechanism is used, then, to update the proxies with actual values with a specified periodicity or on demand.

At a high level of abstraction of the AADL model, the state variable proxy notion can be encapsulated in the protocol used by the telemetry (*SpaceLink*) bus component. It is the responsibility of the protocol to distribute the state to the *out data ports* of components to other components. For data port connections across the *SpaceLink* bus, a different protocol is used to provide the desired

caching strategy of the state variable proxy. The application model is agnostic to this proxy/caching scheme.

If it is desirable to explicitly model the proxy scheme, we can do so in two ways. We can model an implementation of the proxy/caching protocol of the telemetry bus component as a separate AADL model that is associated with the *SpaceLink* bus by property. This property is interpreted by the instance model generator to refine the bus abstraction by its implementation. Alternatively, we can model the proxies explicitly as application components (i.e., as threads that receive the original data port content by executing at a specified rate and make it available locally). In this case, users need to modify the model by inserting or removing the proxies as components that are migrated between flight system and ground.

3.5 Model Organization

AADL packages are used to organize the model space. We place all packages making up the MDS reference architecture in one project in the OSATE tool environment. We define the MDS reference architecture as a collection of packages as illustrated in Figure 9.

```
MDS_Reference_Architecture
  aadl
    packages
      CompleteMDSSystem.aadl
      ControlSoftware.aadl
      ExecutionHardware.aadl
      HardwareAdapters.aadl
      MDSControlSystem.aadl
      MDSData.aadl
      SystemUnderControl.aadl
      ValueHistories.aadl
    propertysets
      RateGroups.aadl
```
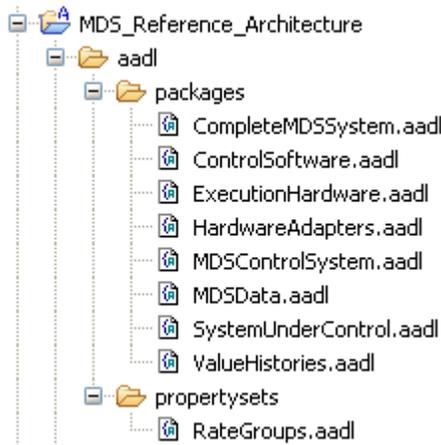
Figure 9:   Packages of the MDS Reference Architecture

The package *MDSData* contains all declarations of port group types and data component types used in the AADL model of MDS. The data component types are used in data port declarations to specify the data type of the data communicated through these ports. The *ValueHistories* package provides declarations for value history modeling. It is used by the hardware adapters and the estimators. The *SystemUnderControl* package contains the system declaration for the system under control. The *HardwareAdapters* package contains the systems representing the sensor adapters and the actuator adapters. The *MDSControlSystem* package contains the MDS control system, while the components of the MDS control system (i.e., the estimators, controllers, goal executive, goal monitor, and goal elaborator, are declared in the *ControlSoftware* package).

The MDS reference architecture will be instantiated to represent a specific MDS system (e.g., a *CameraSystem*) in a second set of packages in a separate OSATE project (see Figure 10). This allows multiple MDS instances to be developed independently.
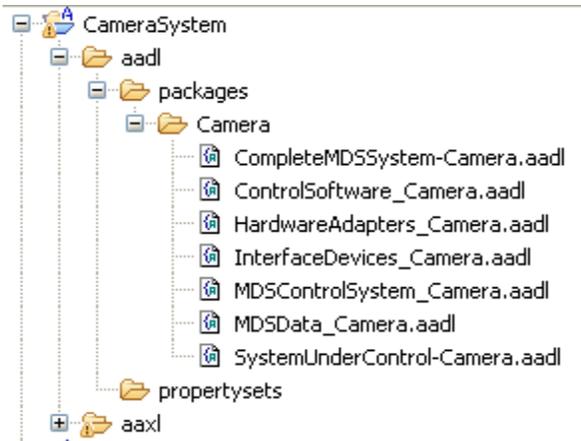
Figure 10: Packages of an MDS instance

An excerpt from the *ControlSoftware* package is shown in Figure 11. The elements of the computer platform are declared in the *ExecutionHardware* package. Finally, the top-level system is declared in the *CompleteMDSSystem* package.

```
package ControlSoftware
public
  -- this type is refined for a MDS instance by refining the
  -- classifiers of the features to be instance specific
  thread group controller
    features
      StateEstimatesIn: port group MDSData::StateEstimatesIn;
      EstimateHistoryIn: port group ValueHisto-
ries::EstimateHistoryInv;
      ControlGoalsIn: port group MDSData::XgoalsIn;
      CommandsOut: port group MDSData::CommandsOut;
  end controller;

  thread group implementation controller.basic
  end controller.basic;
```

Figure 11: Example Package of MDS Reference Architecture

In addition to the packages, we have defined a property set in the MDS reference architecture project. This property set defines properties for modeling rate groups. Other property sets can be added to introduce additional properties that are specific to the MDS architecture.

The MDS reference architecture is refined in a set of nested packages, as shown in Figure 12. Individual components of the MDS reference architecture are refined by making use of the *extends* and *refines* concepts of AADL. The extended port group types, component types, or component implementations refine previously declared features and subcomponents, and they can add subcomponents or features. We will declare port group type extensions that fill in the details of the port groups defined in the reference architecture (e.g., the specific set of estimated states). We will declare component type extensions that refine the classifiers of their features to the instance-specific port group and component classifiers. We will declare component implementation extensions that introduce specific instances of estimators, controllers, and so on through subcomponent declarations.

```
package ControlSoftware::Camera
public
  thread group controller
    extends ControlSoftware::controller
    features
      StateEstimatesIn: refined to port group
              MDSData::Camera::StateEstimatesIn;
      EstimateHistoryIn: refined to port group
            ValueHsitories::Camera:: EstimateHistoryIn;
      CommandsOut: refined to port group
            MDSData::Camera::CommandsOut;
      ControlGoalsIn: port group
            MDSData::Camera::ControlGoalsIn;
    flows
      ControlFlow: flow path StateEstimatesIn -> CommandsOut;
  end controller;

  thread group implementation controller.camera
    subcomponents
      HeaterController: thread HeaterController;
      HeaterSwitchEstimator: thread HeaterSwitchEstimator;
```

Figure 12: Example Package of an MDS Instance

3.6 Operating System Thread Model

Hardware adapter, estimator, controller, planner, goal executive, and goal monitor are represented by logical threads, each with an execution rate, a deadline, and a worst-case execution time. Some of this functionality may be distributed between flight system and ground or may be distributed within the flight system or ground system. The latter distribution may occur due to a multi-processor configuration or in anticipation of using multi-core chip architectures in a spacecraft.

Distribution decisions regarding ground or flight system are localized to changes in processor binding property values in the AADL model, unless state variable proxies are modeled explicitly as part of the application system. The collection of logical threads bound to the ground processor or the flight processor is then grouped into rate groups. Each member of a rate group is executed by an operating system thread at the period of the rate group. Note that such rate group optimization must take into account execution order requirements between threads of the same rate or of different rates that require data to be communicated mid-frame (i.e., within the same execution cycle).

```
property set RateGroups is
  RateGroups : type enumeration ( EstimatorRategroup,
      ControllerRateGroup, PlanExecutionRateGroup,
      PlanningRateGroup, HWARateGroup);
  AssignedRateGroup : inherit RateGroups::RateGroups
                    applies to (thread, thread group, process, sys-
tem);

end RateGroups;
```

Figure 13: Rate Group Modeling by Properties

Rate group optimizations can be represented within the current version of AADL using the property mechanism. We can introduce a property type *RateGroups* that is an enumeration of rate groups in a particular application and a property to specify the rate group that a thread is assigned to, as illustrated in Figure 13. The enumeration literals are an ordered set.

AADL V2[2] introduces the concept of virtual processor to model hierarchical schedulers. The operating system threads, which execute the tasks of a rate group, act as schedulers that dispatch these tasks as a cyclic executive. Therefore, we represent each of them as a virtual processor to which the application AADL threads are bound. Each of these virtual processors is defined as a subcomponent of a given processor or is defined separately and bound to a processor.

3.7 Binding to Hardware

AADL supports modeling the computer platform of the embedded system. In Figure 14, we illustrate how flight system and ground system computer platforms can be modeled. The flight system consists of a processor, memory, and a flight system bus. In addition, the flight processor has access to a device bus that is also accessible by devices representing the sensors and actuators outside the MDS computer hardware system component. The ground system consists of a processor, memory, and a ground system bus. The two computer platforms are interconnected via a *SpaceLink* bus that represents the downlink between the spacecraft and the ground station.

Without having to model the internal details of the hardware, we can use properties to specify characteristics relevant to the analysis of embedded systems. For example, a processor has specified context switch and cycle times that may have been determined through measurements of the actual hardware or through simulation runs of a VHSIC hardware description language (VHDL) model of the processor.[3] Similarly, the bus components may include not only properties that characterize transmission timing, but also properties that characterize the quality of service of the protocols used by the bus, such as secure and guaranteed delivery.

---

[2]   Version 2 of the AADL standard (AADL V2) was published in January 2009 by SAE International [SAE AADL 2004/2009].

[3]   VHSIC stands for very-high-speed integrated circuit.
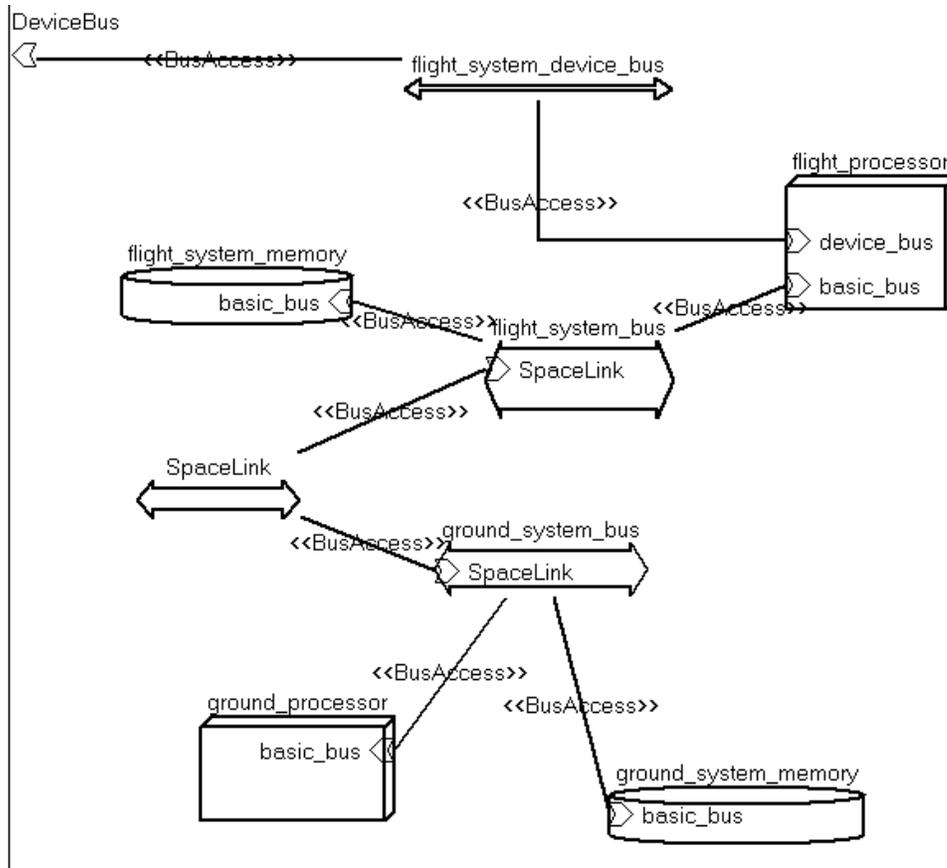
Figure 14: Flight and Ground Processing Systems

The binding of embedded software applications to the computer platform is also accomplished through properties. The *Allowed_Processor_Binding* property places constraints on the binding to processors. The binding may be constrained to a processor type or to a set of processors. Binding constraints are taken into consideration when a resource allocation tool makes its allocation decisions; the *Actual_Processor_Binding* property records the actual binding decisions.

Figure 15 shows the use of *Allowed_Processor_Binding* for the MDS architecture. This property is declared with the top-level system implementation, allowing the property declaration to refer to the processor as the reference value and to the application component to which the property applies.

```
package CompleteMDSSystem::Camera
public
  system CompleteMDSSystem
    extends CompleteMDSSystem::CompleteMDSSystem
  end CompleteMDSSystem;

  system implementation CompleteMDSSystem.Camera
    extends CompleteMDSSystem::CompleteMDSSystem.basic
    subcomponents
      MDSControlSystem: refined to process
          MDSControlSystem::Camera::MDSControlSystem.camera;
      ControlledMDSSystem: refined to system
          SystemUnderControl::Camera::system_under_control.camera;
```

```
        MDSPlatform: refined to system
            ExecutionHardware::Camera::MDSHardware.camera;
    flows
      TemperatureResponse: end to end flow
            MDSSystemUnderControl.Tempflow -> SystemtoControllerConn -
>
            MDSControlSystem.ControlFlow -> ControllertoSystemConn ->
            MDSSystemUnderControl.HeaterCmdFlow
                { Latency => 50 ms;};
    properties
      Allowed_Processor_Binding =>
                reference mdsplatform.ground_processor applies to
MDSControlSystem.OperatorConsole;
      Allowed_Processor_Binding =>
                reference mdsplatform.ground_processor applies to
MDSControlSystem.GoalElaborator;
      Allowed_Processor_Binding =>
                reference mdsplatform.flight_processor applies to
MDSControlSystem.GoalExecutive;
      Allowed_Processor_Binding =>
                reference mdsplatform.flight_processor applies to
MDSControlSystem.StateEstimation;
      Allowed_Processor_Binding =>
                reference mdsplatform.flight_processor applies to
MDSControlSystem.DeviceControl;
      Allowed_Processor_Binding =>
                reference mdsplatform.flight_processor applies to
MDSSystemUnderControl.HardwareAdapters;
```

Figure 15: Modeling of Processor Bindings

4 An MDS Instance

In this section, we show how AADL is used to refine the reference architecture model of MDS (defined in the previous section) into an instance of MDS for a specific system, the platform-mounted camera temperature and pointing control system. Sections 5 and 6 report subsequent analysis supported by AADL conducted as part of this case study.

4.1 The Heated Camera System

This example, shown in Figure 16, is taken from the MDS tutorial "State Analysis for Software Engineers: Model-Based Systems and Software Engineering" [Bennett 2006].
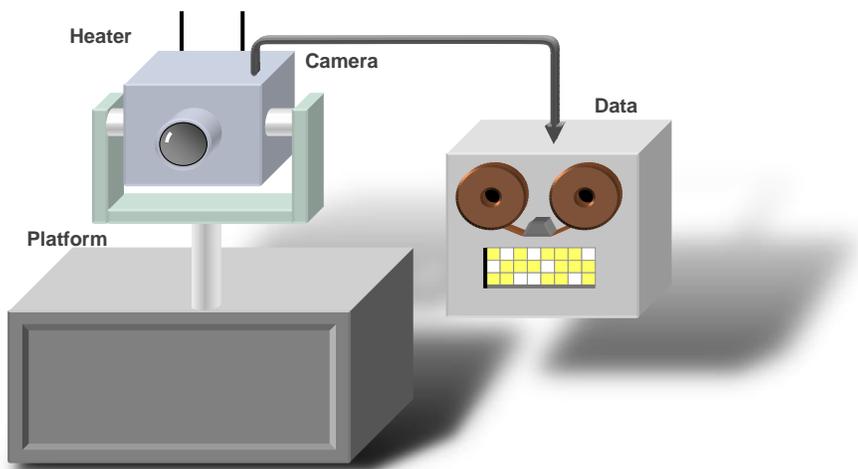
Figure 16: Platform-Mounted Camera [Bennett 2006]

In the example, the main control loop is a flow of the temperature signal from the temperature sensor (modeled as an AADL device), through the control system, and terminating in the actuator for the power switch of the camera heater (also modeled as an AADL device). The control sensing and actuation and relevant interfaces are shown in Figure 17.
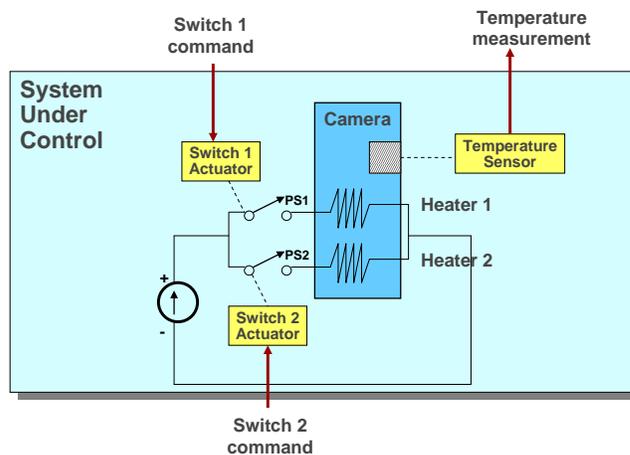
Figure 17: Fault-Tolerant Camera Heater Control System

4.2 The AADL Model of the Heated Camera System Hardware
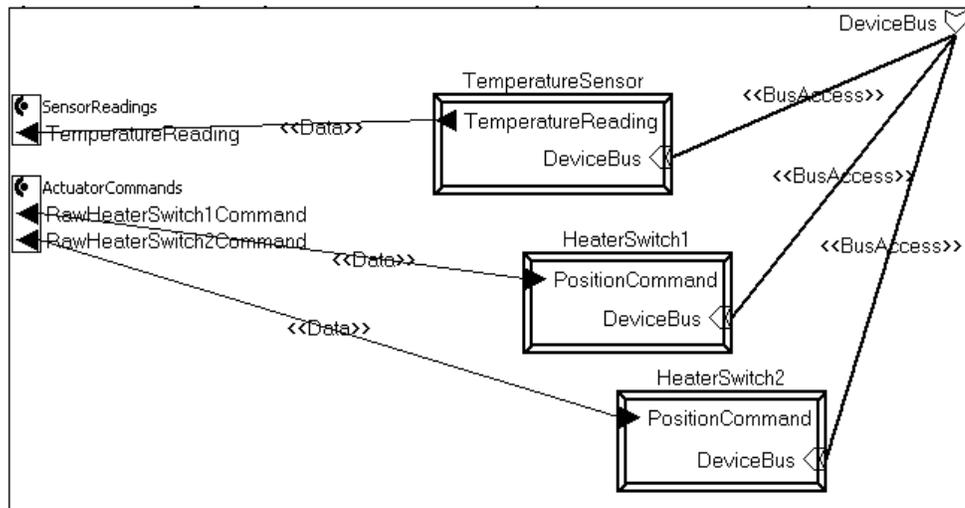
Figure 18: AADL Representation of the Camera Hardware System under Control

The AADL graphical representation of the camera hardware that is the system under control is shown in Figure 18. We define the camera hardware by refining the *SystemUnderControl* component defined in the MDS reference architecture. We add the temperature sensor and the heater switches as separate devices. These devices are defined in a separate package called *InterfaceDevices_Camera* as shown in Figure 10 on page 16. These devices are physically connected to the device bus and provide a logical connection to the MDS application through the refined *SensorReadings* and *ActuatorCommands* port groups. These port groups have been refined to define the individual data ports used for communicating measurements and commands. The refinement of the *SensorReadings* port group type is illustrated in Figure 19.

```
package MDSData::Camera
public
  port group SensorReadingsOutput
    extends MDSData::SensorReadingsOutput
    features
      TemperatureReading: out data port TemperatureReading;
  end SensorReadingsOutput;
```

Figure 19: Refinement of the Sensor Readings Port Group

4.3 The Heated Camera Control System

We refine each of the control system components of the MDS reference architecture. The sensor and actuator hardware adapter components are refined by defining an adapter thread for each of the adapters for the heated camera system. Similarly, we refine the estimator, controller, goal executive, and goal monitor components with threads. The estimator and controller components and their interactions are defined in the MDS documentation in a collaboration diagram shown in Figure 20.
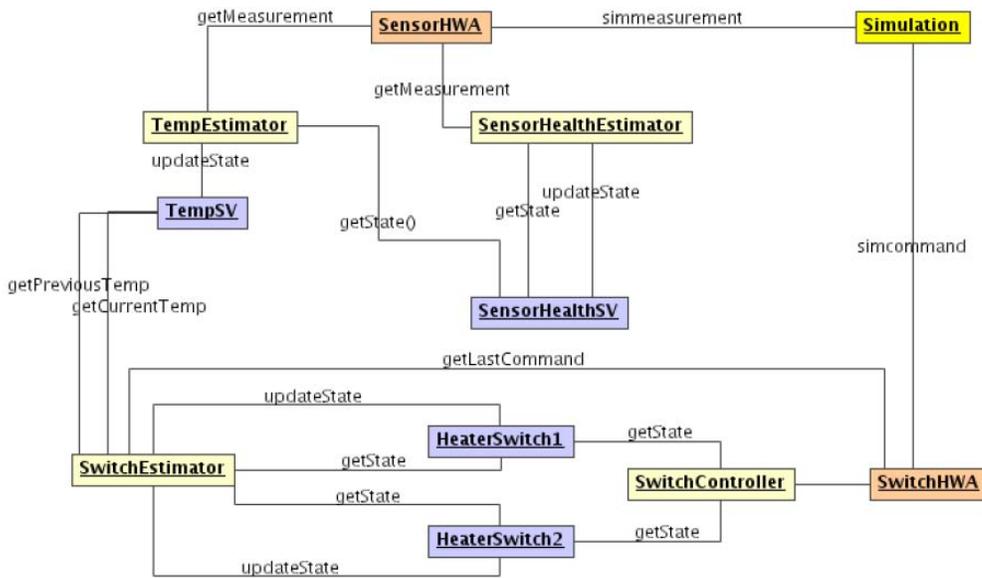
Figure 20: Collaboration Diagram of Camera Heater Control

The equivalent AADL model is shown in Figure 21 and Figure 22. We will take advantage of the information in the collaboration diagram as to whether a component accesses the current value, the previous value, or the value history, and we will represent them through different port connections.

The estimators make use of temperature measurements, temperature sensor health state, and heater switch state. The estimated state is represented by *out data ports* and is available to other estimators, shown as a connection to the respective *in data port* and to controllers via the *StateEstimates* port group. This port group has been refined for the camera heater system to define all the state variables updated by estimators as *out data ports*.
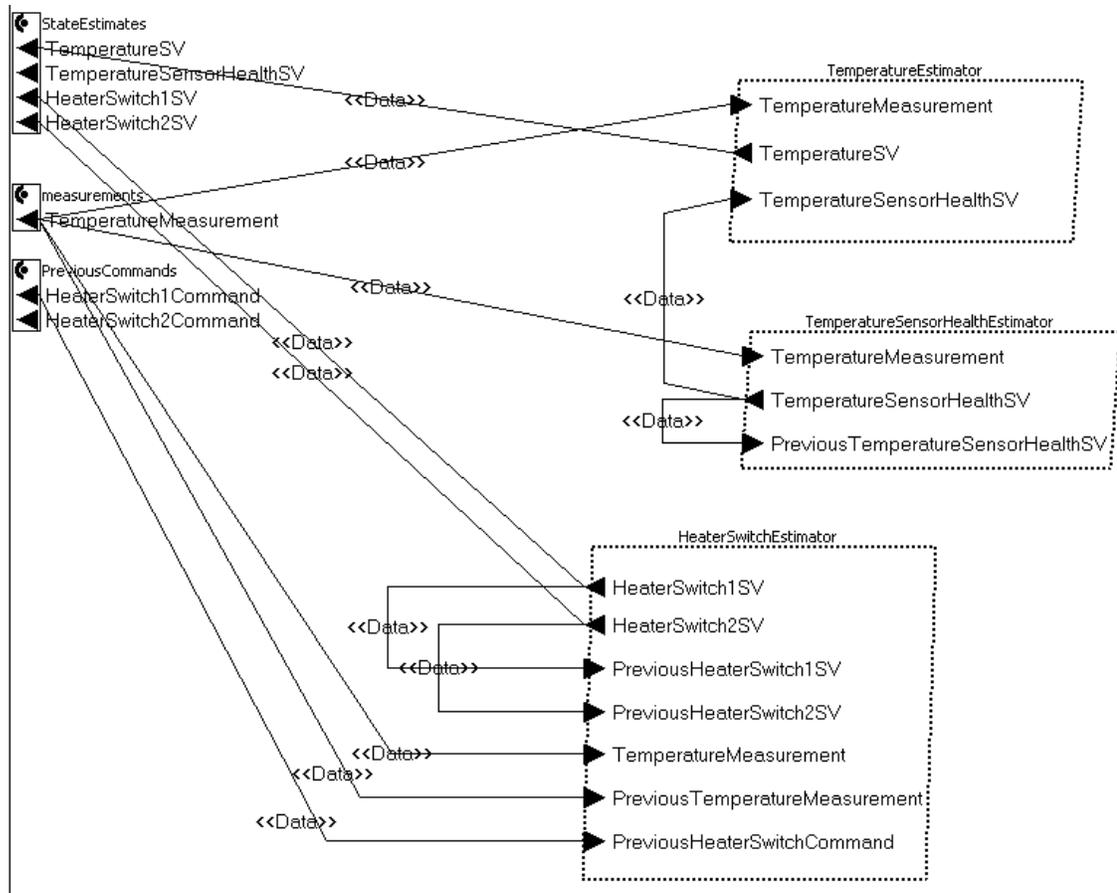
Figure 21: Camera Heater Estimators

The *getState* actions on the state variables are mapped into data port connections from the *out data port* of the provider of values to a state variable. Access to the current value is represented by an immediate data port connection, while access to the previous value is represented by a delayed data port connection. The fact that a connection is immediate or delayed is a property of the connection and visible in the properties viewer. In the textual representation of the model, it is expressed through the symbols -> and ->> respectively.

The heater switch controller takes heater goals as input and produces heater switch commands. It takes the estimated state of the heater switches into account to determine whether the heater switches are functional.
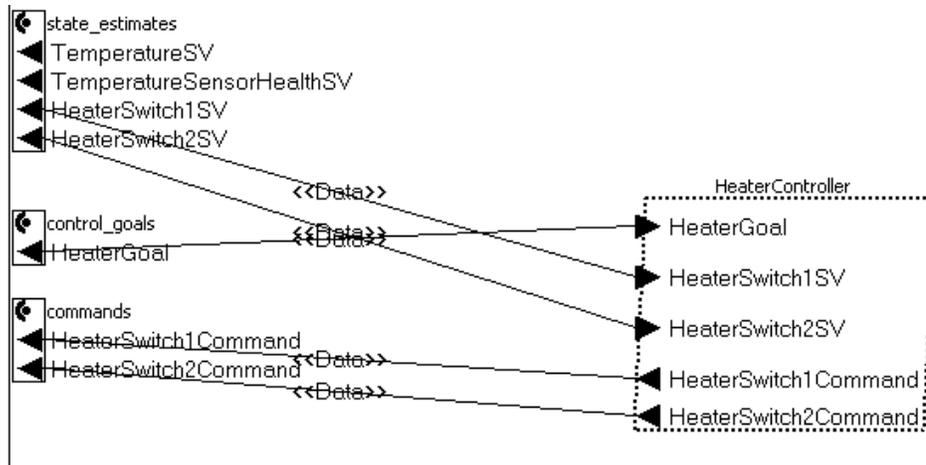
Figure 22: Camera Heater Controller

4.4 The Refined Top-Level System

For some of the system components of the MDS reference architecture, the refinement into the heated camera instance simply involves refining the classifiers from the generic classifiers of the reference architecture model to the heated camera system specific classifiers. This refining is illustrated in Figure 23 for the top-level system description by specifying the camera implementation of the MDS control system, the controlled MDS system, and the MDS platform.

```
package CompleteMDSSystem::Camera
public
  system CompleteMDSSystem
    extends CompleteMDSSystem::CompleteMDSSystem
  end CompleteMDSSystem;

  system implementation CompleteMDSSystem.Camera
    extends CompleteMDSSystem::CompleteMDSSystem.basic
    subcomponents
      MDSControlSystem: refined to
    process MDSControlSystem::Camera::MDSControlSystem.camera;
      ControlledMDSSystem: refined to
    system SystemUnderControl::Camera::system_under_control.camera;
      MDSPlatform: refined to
    system ExecutionHardware::Camera::MDSHardware.camera;
```

Figure 23: Refinement of the Top-level System

4.5 System Analysis

Although the main focus of AADL is the embedded software system architecture, AADL also supports modeling the physical system under control. In Section 4.2 we used the AADL device concept to capture physical entities such as sensors and actuators (switches). Devices can also represent physical components such as an engine with ports representing sensor output and actuator input. These physical components have mass and the system may have constraints regarding its maximum mass.

A system under control may also contain components that represent physical resources, such as electrical power and hydraulic power. We use the AADL bus components to represent a physical resource such as electrical power or hydraulic pressure. These resources are supplied to physical subsystems, such as the heater, through bus access connections. In other words, these bus access connections represent fuel lines and hydraulic hoses.

In the next two sections, we illustrate how AADL can be used to perform a coarse-grained analysis, which can be refined with the AADL model, of physical resources or resource consumption issues that are addressed as part of the goal network representation.

4.5.1   Mass and Weight Analysis

We have introduced three mass related properties in order to perform mass analysis.

1. *SEI::MassLimit*: the maximum acceptable mass for a system. This property can be used for the system under control and for the computer hardware system. The mass is expressed in units of *kg* and takes a real value.

2. *SEI::NetMass*: the net mass of a system. This is the net mass of an empty cabinet or a board without its mounted components. The mass is expressed in units of *kg* and takes a real value.

3. *SEI::GrossMass*: the gross mass of a system. This is the net mass of a system (component) plus the sum of the gross mass of its subcomponents. For example, this is the mass of a cabinet plus the mass of the boards including their mounted parts. The mass is expressed in units of *kg* and takes a real value.

It is expected that if both the net mass and the gross mass are specified for a component, their values would be consistent (i.e., the net mass of the component plus the gross mass or its equivalent in the form of net mass plus its subcomponents' gross mass must be the same). For leaves (nodes) in the component hierarchy, the net mass and gross mass are expected to be identical. It is expected that a component's gross mass does not exceed its mass limit.

The mass analysis () can be invoked on an instance model of the MDS. For example, we can instantiate the reference model and associate a mass with the physical system under control as a whole as well as the computer platform. As the reference architecture model is refined into a model of a specific system and the physical system under control is populated with parts, we can revisit this mass analysis for the complete system. We can also limit the mass analysis to an instance of the computer platform by creating an instance model of the system implementation that represents the computer hardware.

We support mass analysis in two forms:

*   Mass analysis through an OSATE analysis plug-in that totals the net mass property value of components (systems and buses) in the instance model, relates them to the gross mass values, and compares them against mass limit values.

*   Mass analysis by exporting the mass information into an Excel-compatible CSV file with the analysis performed in Excel.

In order to be able to perform the mass analysis we must assign mass property values to the system type or system implementation declaration of the MDS and its subsystems (i.e., the system

type or system implementation declarations of the system under control and the computer hardware). In addition we can assign mass properties to the bus types representing physical resource containers and the bus access connections that represent physical connectors to such a physical resource.

In the context of a spacecraft system, it may be necessary to distinguish between mass and weight. Weight is proportional to mass in a uniform gravitational field. Some control applications such as that of a rover may utilize weight in its stability calculations. Such characteristics can be captured through additional properties; AADL supports the introduction of user-defined properties to address this need.

4.5.2 Power Draw Analysis

In this section we illustrate modeling of consumable and renewable resources in AADL for a coarse-grained resource analysis. We use electrical power as the example. The power is expressed in units of milliwatts (mW), watts (W), and kilowatts (KW). We have three power related properties:

1. *SEI::PowerCapacity*—the power capacity provided by a physical system component such as a battery. This property is used on bus or system components to indicate the amount of power available to power consumers. If power suppliers are connected to this component, then their power supply total should correspond to the power capacity. Currently the capacity is specified in terms of watts reflecting a power system with renewable power. For non-renewable resources such as a non-chargeable battery, we may also want to specify the energy capacity in terms of power and time (e.g., kilowatt-hour [KWH]).

2. *SEI::PowerSupply*—the amount of power supplied to a power carrier. This property is used on *requires bus access* features of power producer components, such as a solar panel.

3. *SEI::PowerBudget*—the amount of power consumed by a component. This property is used on *requires bus access* features of power consumer components, such as the heater or a motor.

It is expected that the power capacity of a component corresponds to the sum of the supplied power (i.e., supplier components connected by bus access with a *PowerSupply* property value). It is expected that the sum of the power budgets of connected components does not exceed the capacity or supplied power.

Figure 24 illustrates the use of an AADL bus of type *PowerSupply* to represent an electrical power resource. The power supply has a power capacity property indicating its available power. The figure also shows three power consumers that are connected to the power supply through bus access connections. Their *requires bus access* features have property values indicating their power budgets.

A bus instance that represents a power resource may have a *requires bus access* feature itself. This indicates that this bus can be connected via bus access to another bus to draw on that buses power capacity. The *PowerBudget* property value indicates the amount of power drawn. In other words, the capacity of a bus is the sum of *PowerSupply* values of connected components plus the sum of the *PowerBudget* values of its own bus access connections.
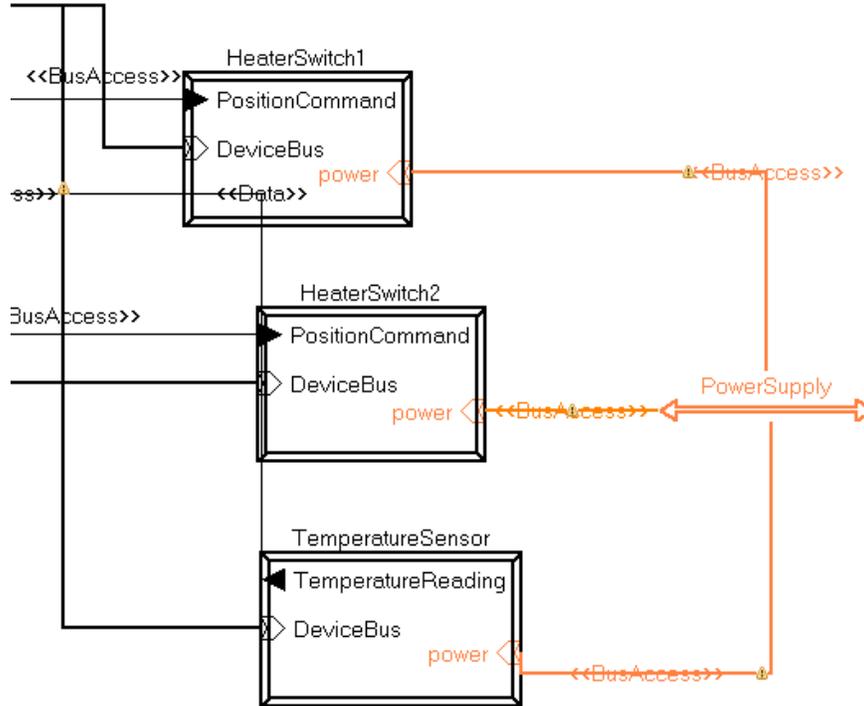
Figure 24: Power Supply as a Physical Resource

The power analysis ( ) can be invoked on an instance of the MDS. It can be performed on an initial architecture model and then revisited as the model is refined to a greater level of detail.

We support power analysis in these forms:

- through an OSATE analysis plug-in that totals the power budgets of connected components and compares the total against the power capacity of the power resource (bus). It takes into account *PowerSupply* connections as well as *requires bus access* by the power source.

- by exporting the power information into an Excel-compatible CSV file with the analysis performed in Excel

- by exporting the power information into a format that is understood by the goal network analysis tool of MDS. This allows us to ensure that power-related numbers are used consistently.

Such a power analysis may reveal that the capacity of the bus supplying the power is not sufficient. We can now explore alternatives, such as a power bus with higher capacity, or components with lower power consumption. As we do so, we can immediately analyze the impact on other quality dimensions. For example, a higher capacity battery may increase the mass, and a lower powered processor may provide fewer execution cycles and, thus, be able to handle fewer tasks.

5   Closed Loop Control System

In this section we examine how to best represent the closed loop control system of MDS in AADL. Our starting point, the state-based design of MDS, leads intuitively to an AADL model that represents the state variables as data components accessed by different functions. However, because we are dealing with time-sensitive data with a continuous value range, variation in time is observed as increased noise in the data, which can negatively affect the stability of controllers.

Cervin performed a case study of different scheduling algorithms on the stability of controllers [Cervin 2006]. Figure 25 shows the effect of those schedulers. The flattest curve shows a scheduler where first input for all tasks is performed, then tasks compute according to their priority, and finally output is made available. This corresponds to the AADL execution model of freezing input at task dispatch time.
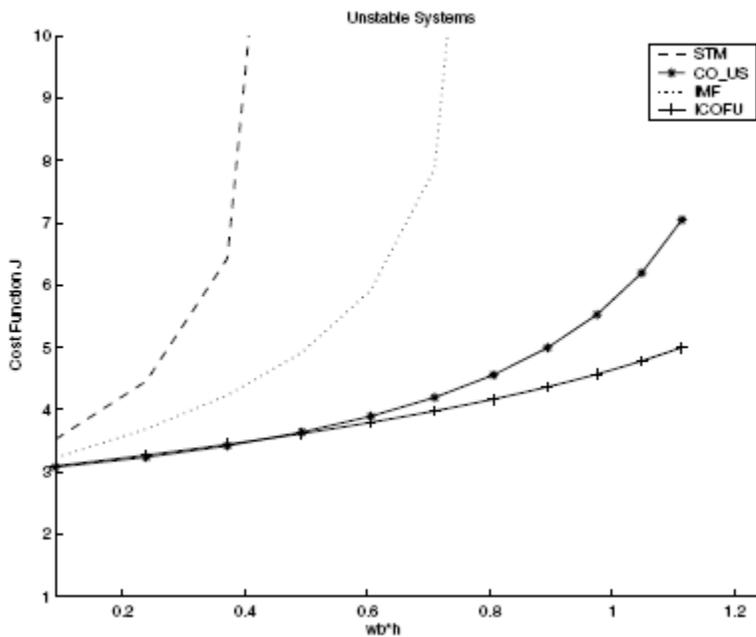
Figure 25: Impact of Latency Jitter on Controller Stability

Latency is also sensitive to the order in which the sender and receiver communicate their data, (i.e., the order in which state variables are written and read or in which send and receive operations are performed). In other words, the execution order of the tasks drives the information flow and its latency. Preemptive scheduling and concurrent execution of tasks on different processor cores or different processors contribute to frame-level jitter.

In this section we first discuss a flow-oriented representation of the closed loop control system in AADL that still reflects the state-based design approach of MDS. Then we illustrate how this model can be used in performing end-to-end latency analysis.

5.1 MDS State Variables and Data Flow

Figure 2 on page 7 illustrates the separation of what information is communicated between components and how it is communicated through state variables in the MDS architecture. The Unified Modeling Language (UML) class diagram shown in Figure 26 provides greater detail about the content of MDS data objects; their relationships among MDS estimator, controller, and hardware adapter software elements; and their realization through state variables. State variables are containers that

- store and make available data values of data streams
- maintain a value history and make that history available through state functions

Furthermore, MDS provides functionality for state-variable value history management, such as data compression, and its transfer between flight system and ground system (shown in Figure 8).
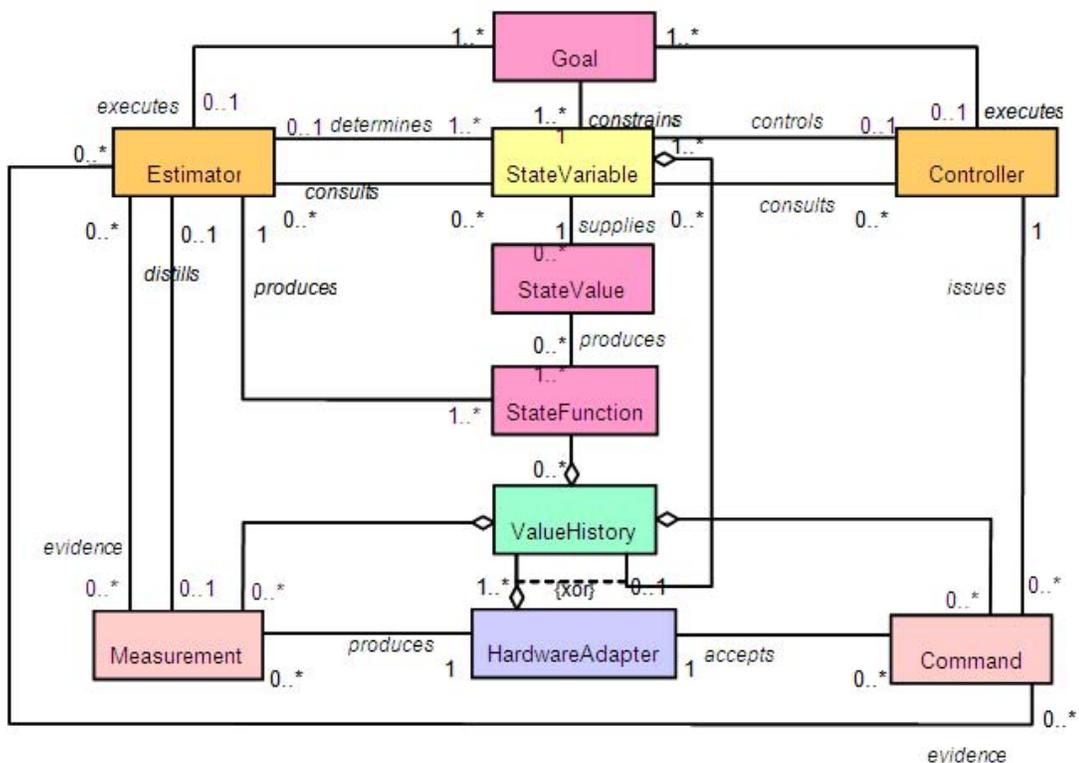
Figure 26: MDS Software [Bennett 2006]

Given the state-based design approach of MDS, it is logical to represent the state variables as data components. These data components are then accessed through data access connections, as illustrated in the left box of Figure 27. However, they are part of the data stream of the control loop from the sensors, their adapters, and the state estimators to the controllers, actuator adapters, and the actuators themselves. The flow of information has to be inferred from the access rights to the data components. MDS requires that only one functional unit write to the state variable (i.e., only one component has write access). There can be multiple readers of state variables (i.e., multiple access connections with read access). A visualization of this intended flow is shown in the top right box of Figure 27.

This flow-oriented view can be mapped into a port and connection model. The functional unit producing the values for a state variable has an output data port that makes the most recent value accessible. This value is then routed to the consumers by port connections as shown at the bottom right of Figure 27. This allows us to explicitly indicate whether data is intended to be communicated across frames or within the same frame. In the shared variable representation this information is implicit in the order in which the write and read operations occur in every frame. Changes in schedulers or use of multiple processors can result in non-deterministic write/read order, which in turn results in frame-level latency jitter of the sampled data. This introduces software-induced noise into the data, which may affect the stability to the control system.
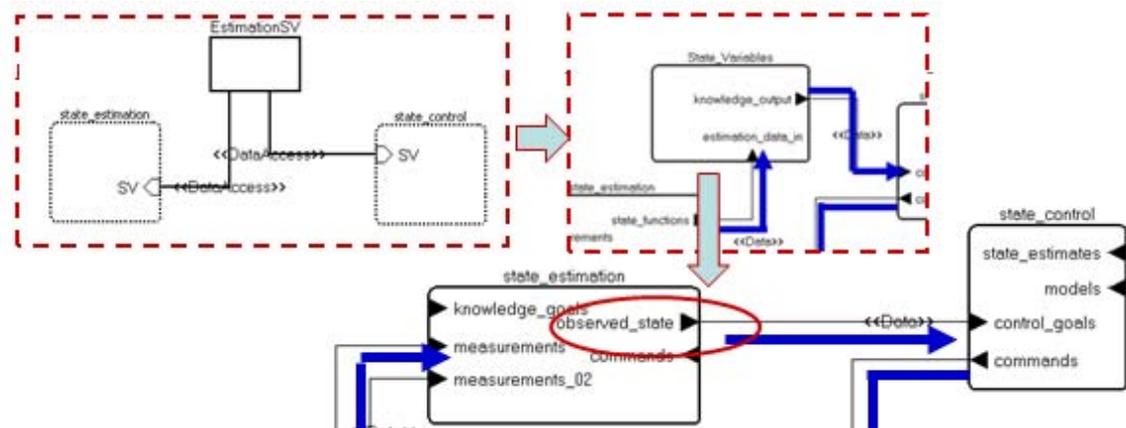
Figure 27: From State Variable to Port-Based Flow

5.2 Representing the Control Loop Data Stream

States allow for goal-oriented task modeling and for separation of state estimation and control concerns. Measurements from sensors represent observations of characteristics of the physical plant state, which are used by estimators to determine an accurate estimate of the physical state. This state is used together with desired system states (i.e., goals) to change the state of the physical system.

In order to explicitly model the timing assumptions of the information flow, we utilize data ports and immediate, delayed, and sampling connections for data ports to indicate whether mid-frame and phase-delayed data flow is assumed to occur or non-deterministic sampling is acceptable. We take advantage of the fact that an estimated state is updated by a single estimator (i.e., its *out port* effectively represents the stream of estimated values for an estimated state). This estimated state may be determined from measurements or derived from other states (i.e., a derived state estimator uses the output of other estimators as input). Connections from the estimator *out data ports* to *in data ports* of individual controllers or to estimators responsible for determining derived states reflect the information flow embedded in the state effects diagrams. By declaring data port connections to be immediate connections, we specify that the users of the state expect to see the new data value. When an estimator or controller utilizes a previous state value, we will indicate this by declaring a delayed data port connection.

Immediate and delayed data port connection declarations are used in the refinement of estimators and controller components as part of an MDS architecture instantiation. For example, a sensor

device may have an *out port* that provides raw temperature readings. This value is fed to a sensor adapter that normalizes the temperature reading into a temperature measurement. The measurement is then passed to a temperature estimator, which uses the measurement as evidence to determine the current temperature. The estimator may take into account other measurements and state information (e.g., temperature sensor failure).

Similarly, the commands for an actuator are the responsibility of a single controller (i.e., its *out port* effectively represents the stream of control commands). This fact allows us to use the *out data ports* of the estimators to represent the estimated state as a data stream and the *out data ports* of the controllers to represent the stream of control commands.

5.3 Flow Latency Analysis

In this section, we demonstrate the capability of AADL to model end-to-end flows and utilize these specifications to perform end-to-end latency analysis. From a control engineer's perspective, end-to-end latency consists of

- processing latency to perform the control computation

- sampling latency due to over- and under-sampling

- transmission latency of the signal from the sensor and the signal to the actuator over physical connections

When the control system is implemented as software, we have a number of additional contributors to end-to-end latency including the sharing of processor and network resources, preemptive scheduling, blocking due to mutually exclusive access to shared logical resources such as shared data areas, use of partitioned architectures, and rate group optimization.

The SEI has developed a latency analysis framework around AADL models that utilizes end-to-end flow specifications as well as knowledge about the execution of the control application as a collection of application threads executing at a given rate and communicating their results via different communication mechanisms [Feiler 2007, Feiler 2008]. We will utilize an implementation of the flow latency analysis capability in OSATE in this section.

As part of our initial case study, the flow latency analysis explores the end-to-end latency for the Heated Camera System MDS instance described in Section 4.

5.3.1   End-To-End Flow Specification

TemperatureResponse, an end-to-end flow specification, is defined to account for a signal from the temperature sensor through the control system to the switch actuator device. This measure records the time from a switching threshold temperature being sensed to the switch actuator receiving a command to turn the heater on or off. Its path is defined as an end-to-end flow originating at the *TemperatureSensor* device within the camera hardware (*SystemUnderControl*), moving through a path in the *MDSControlSystem* for the camera, and ending in the *HeaterSwitch* device within the camera hardware (*SystemUnderControl*).

AADL supports flow specifications for individual components, which allows a user to specify the flow characteristics through a component without having to expose the internal details of the component. Within a component implementation, flow specifications are detailed into flow im-

plementations that indicate how the flows are realized through the subcomponents making up the component. These flow specifications may represent flow sources (the flow starts within the component), flow paths (the flow goes from an in port to an out port), and flow sinks (the flow ends within the component).

5.3.2   Flow Specifications for Sensors and Actuators

Given flow specifications for a set of components, we can specify an end-to-end flow through those components. In our example, we will specify the end-to-end flow *TemperatureResponse* by declaring it to start with the flow source *TempFlow* of the controlled MDS system, via connection *SystemtoControllerConn* through the flow path *ControlFlow* of the MDS control system, and via connection *ControllertoSystemConn* terminate in the flow sink *HeaterCmdFlow* of the controlled MDS system. This end-to-end flow specification is shown in Figure 28. This specification includes a latency property to indicate that the expected latency for the end-to-end flow is to be 50 ms. *This value is an arbitrary value selected for illustrative purposes in the analysis example shown in Section 5.4.*

```
package CompleteMDSSystem::Camera
public
  system CompleteMDSSystem
    extends CompleteMDSSystem::CompleteMDSSystem
  end CompleteMDSSystem;

  system implementation CompleteMDSSystem.Camera
    extends CompleteMDSSystem::CompleteMDSSystem.basic
    subcomponents
      MDSControlSystem: refined to
        process MDSControlSystem::Camera::MDSControlSystem.camera;
      MDSSystemUnderControl: refined to
        system SystemUnderCon-
trol::Camera::systemundercontrol.camera;
      MDSPlatform: refined to
        system ExecutionHardware::Camera::MDSHardware.camera;
    flows
      TemperatureResponse: end to end flow
        MDSSystemUnderControl.Tempflow -> SystemtoControllerConn ->
        MDSControlSystem.ControlFlow -> ControllertoSystemConn ->
        MDSSystemUnderControl.HeaterCmdFlow { Latency => 50 ms;};
```

Figure 28: End-to-End Flow Specification

The camera system under control has two flow specifications: a flow source *TempFlow* from the temperature sensor and a flow sink *HeaterCmdFlow* to the heater switch, as shown in Figure 29. Each flow specification is refined in the camera control system implementation into a flow implementation. The flow implementation for the flow source indicates that the flow source starts within the *TemperatureSensor* device and goes through the temperature sensor adapter. The flow implementation for the flow sink indicates that the flow sink goes through the heater switch adapter ending in the *HeaterSwitch* device.

```
package SystemUnderControl::Camera

public
  system SystemUnderControl
    extends SystemUnderControl::system_under_control
    features
      MeasurementsOut: refined to
          port group MDSData::Camera::MeasurementsOutput;
      CommandsIn: refined to
          port group MDSData::Camera::CommandIn;
    flows
      Tempflow: flow source MeasurementsOut;
      HeaterCmdFlow: flow sink CommandsIn;
  end SystemUnderControl;
```

Figure 29: Flow Specifications in MDS System Under Control

5.3.3   Flow through the Control System

The MDS control system for the heated camera system has a flow path specification, which represents the flow through the various processing steps through the control system. This flow specification, called *ControlFlow* (see Figure 30), goes through the sensor adapters, the state estimation, the device control, and the actuator adapters. Within the *StateEstimation* component, the flow is further refined to go through the *TemperatureEstimator*. Similarly, within the *DeviceControl* component, the flow is further refined to go through the *HeaterController*.

```
package MDSControlSystem::Camera
public
  process MDSControlSystem
   extends MDSControlSystem::MDSControlSystem
   features
    MeasurementsIn: refined to port group
        MDSData::Camera::MeasurementsIn;
    CommandsOut: refined to port group MDSData::Camera::CommandsOut;
   flows
    ControlFlow: flow path MeasurementsIn -> CommandsOut;
  end MDSControlSystem;
  process implementation MDSControlSystem.camera
   extends MDSControlSystem::MDSControlSystem.basic
   subcomponents
    GoalPlanner: refined to thread group
            ControlSoftware::Camera::GoalPlanner.basic;
    GoalExecutive: refined to thread group
            ControlSoftware::Camera::GoalExecutive.camera;
    GoalMonitor: refined to thread group
            ControlSoftware::Camera::XGoalMonitor.camera;
    StateEstimation: refined to thread group
            ControlSoftware::Camera::estimator.camera;
    StateControl: refined to thread group
            ControlSoftware::Camera::controller.camera;
   flows
    ControlFlow: flow path MeasurementsIn ->
            Connection1 -> StateEstimation.StateFlow -> Connection5
            -> StateControl.ControlFlow -> Connection2 ->
            CommandsOut;
```

```
    end MDSControlSystem.camera;
end MDSControlSystem::Camera;
```

Figure 30: Control Flow Path Through the Control System

5.3.4   Worst-Case Latency Analysis of a Flow

End-to-end latency of a data flow is determined by several factors:

- processing latency— the amount of time it takes to perform a function. For example, the processing latency of a sensor is the time from the detection of a signal to the corresponding event or message being available at its output. In case of a function realized in software, processing time is the amount of time it takes to compute the function. This time may be bounded by its worst-case execution time, a value used in scheduling analysis to determine schedulability.

- preemption latency— occurs when tasks share a resource. For example, multiple tasks may execute on the same processor, or tasks may require exclusive access to a shared data area. Typically a deadline is specified for a task to indicate the latest time it is expected to complete its execution, since its dispatch. In essence, the deadline represents the worst-case sum of processing time and preemption time.

- communication latency— the amount of time it takes for a signal to travel between application components. This latency may be the time for a signal to physically travel between devices or for data and events to be transmitted via shared memory, a bus, or network. It includes overhead imposed by protocols used to perform the transfer and delay due to resource contention.

- sampling latency— the time delay due to a task reading its input and performing its computation at a specified rate. The maximum latency contribution due to sampling is the period of the recipient.

The end-to-end flow defined in Section 5.3.1 is illustrated as a flattened model in Figure 31. Each element of the end-to-end flow can contribute latency. The OSATE latency analysis tool calculates the worst-case end-to-end latency by first taking into account the latency within the sensor device, which is specified as part of the flow source specification of the sensor. It then accounts for the sensor adapter periodically sampling the sensor output (sampling latency). Then the tool considers the sequence of immediate connections to determine the cumulative latency to be added. Next, the sampling activity of the actuator adapter is considered, and, finally, the latency of the heater switch actuator is added in. If this worst-case end-to-end latency exceeds the expected latency, the analysis tool provides an error message as illustrated in Figure 32. Note that illustrative values were used for this model and the results are not indicative of the results for any existing MDS implementation.
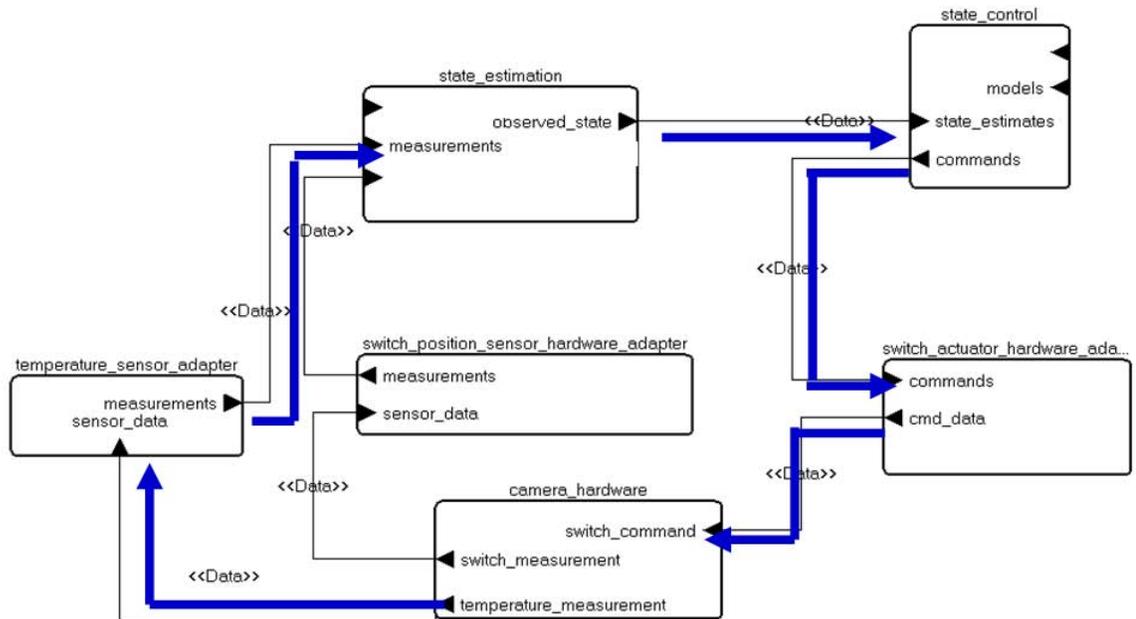
Figure 31: Flattened End-to-end Flow Model
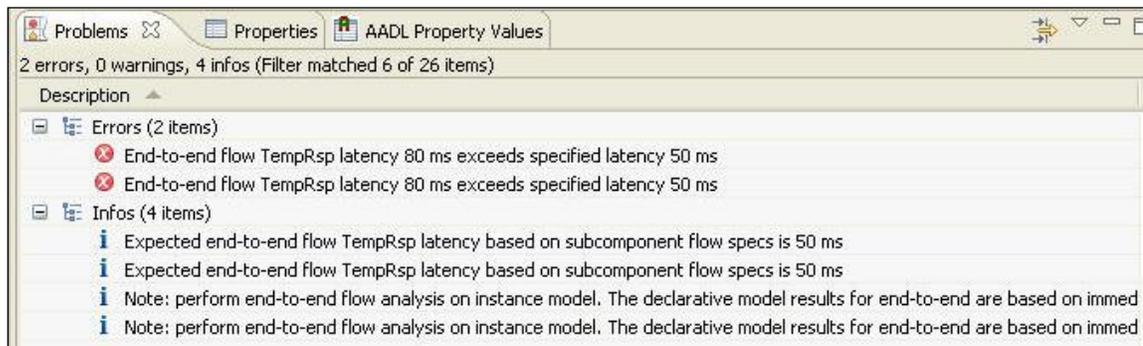
Figure 32: Representative Flow Analysis Output with a Specification Violation

The flow latency analysis capability of the OSATE toolset can be used to investigate the impact on the end-to-end latency of ground-to-flight system migration and vice versa. It can be extended to investigate whether critical flows that are sensitive to latency variation can handle the latency jitter inherent in the particular implementation of the embedded system (see Section 5.3.5).

5.3.5   Analysis of Latency Jitter

Control systems are modeled as continuous time systems and then transformed into discrete time systems. In discrete time systems, the tasks of a control system are performed at known discrete time intervals (frames). In that context, there are the following types of interaction between tasks:

- mid-frame communication (i.e., the output of one task is made available to another task in the same frame)

- phase-delayed (i.e., the output of a task is available to a task, possibly the same task, at the next frame)

- sampled (i.e., a task reads its input at a specified rate independent of the rate of the task whose output is used as input)

In a control system where tasks execute periodically, a periodic task samples its input stream deterministically if it performs mid-frame or phase-delayed communication. If tasks execute at different rates, however, over- or under-sampling occurs. Mid-frame and phase-delayed communication guarantees that a task consistently over- or under-samples deterministically. For example, if the rates of two communicating tasks are harmonic (i.e., one is twice of the other), then the recipient processes every second element in the data stream.
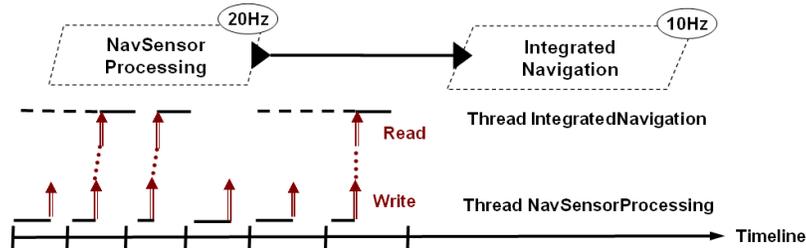
Figure 33: Frame-Level Latency Jitter

However, if deterministic communication between these tasks is not guaranteed, frame-level jitter in latency occurs. For example, if the communication occurs through shared variables and tasks are scheduled preemptively, then the write and read order to those variables is not guaranteed. In this example, the effect is that sampling of the data stream may vary by as much as two frames (see Figure 33).

Deterministic data streaming is important to control systems, as any non-determinism in sampling adds variation to latency and age of data. This variation is in units of frames (the period of the sampling task) and can impact the performance and stability of the control algorithm.

6   Plan Execution and Service Levels

As discussed in Section 2.4, the MDS architecture has a planning layer. This layer has a

- Goal Planner component that is responsible for producing a goal network representation of a mission plan
- Goal Executive that is responsible for achieving the executable goals (*xgoals*) expressed in the goal network by requesting services of different levels from each of the components in the control layer
- Goal Monitor component that is responsible for determining when goals are reached and when it is not feasible to reach a goal and replanning may be necessary

In this section, we focus on how to represent the different levels of service at the control layer and how the *xgoals* are communicated by the planning layer to the control layer. Effectively, the planning layer determines the workload generated by the control layer on the computer platform. In Section 7, we will discuss how goal failure management is mapped into an AADL model representation.

6.1   Modeling of XGoal Execution

Components in the control layer execute periodically to provide feedback control, in order for the system under control to reach a desired goal state. The control system may have algorithms that can handle different operational conditions. One form is a hybrid control system that applies different continuous control algorithms in different physical state regions. Variants of control algorithms may provide different levels of precision in managing the controlled system at the cost of various demands on the computing resources.

In the case of the MDS, the planning layer may ask a control system to be in one of two operational modes: (1) standby, which requires minimal computer resources, or (2) actively driving the system under control to a desired goal state. In our example, the heater controller may be asked to get the camera to be at a certain temperature before it can be used for recording images. Figure 34 illustrates such a goal network for the camera system.

In this scenario, the threads are executing periodically and are informed of new desired state. We can make use of the AADL mode concept to represent the fact that at different times threads should run different variants of algorithms or be in standby, consuming different amounts of execution time in each mode.

Mode-specific property values for *Compute_Entry_Point* can be used to indicate what source code function is to be called in each mode. Specification of computational behavior for each mode can be added through annex subclauses using the AADL Behavior Annex, designed specifically for control system specifications, or a submodel expressed in an existing modeling notation such as Simulink and associated with the AADL model through the *Source_Text* property.

Mode-specific property values for *Compute_Execution_Time* are used to record different resource requirements on the processor. In the next section, we will discuss how this information can be

used to perform workload and scheduling analysis specific to a goal network. Such mode-specific analysis will produce higher fidelity processor resource and scheduling results.

Mode transition conditions can be specified in terms of trigger events such as the arrival on an event port or event data port. By default, the arrival event results in initiating the mode transition at the next hyper period of threads involved in the mode transition.
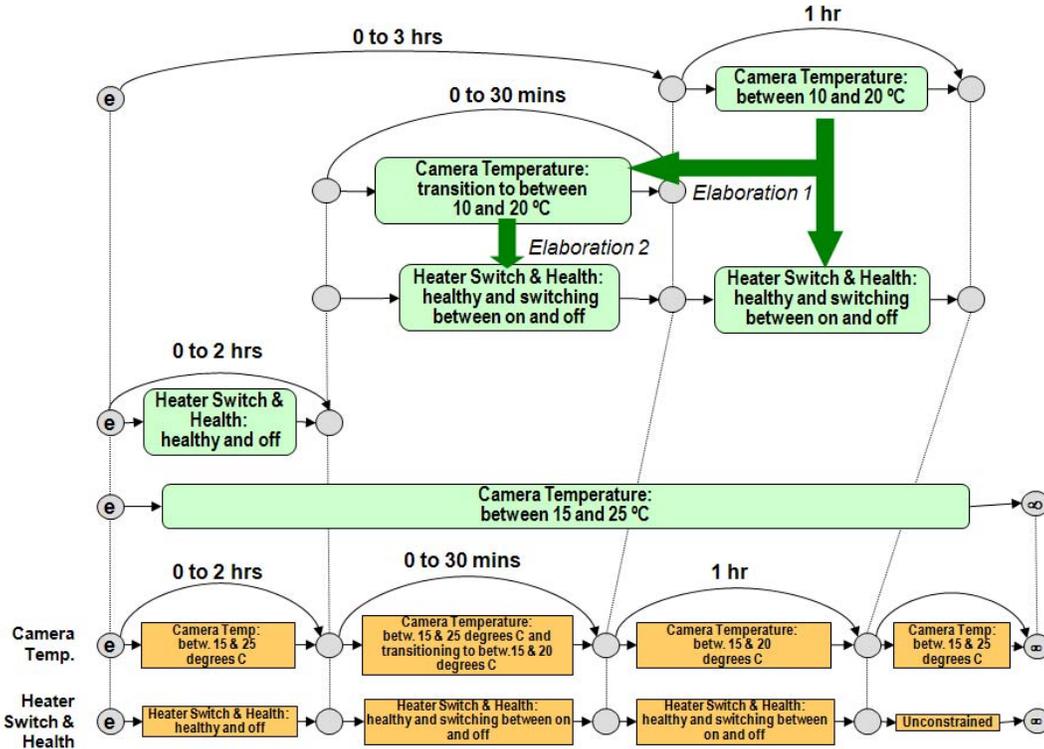
Figure 34: Scheduled Goal Network Drives Control Layer Execution

In our scenario, the supervisor informs the task about the desired level of service, independent of the current level of service. In other words, it sends requests of the form "goto mode x with xgoal parameter y" through separate event data ports. In order to support such behavior, the mode state machine must have a transition from every mode to every mode. In this implementation, we assume that requests for mode transitions do not occur at a higher rate than the rate of execution of the threads. Otherwise, mode transition requests may get lost.

A more intuitive interpretation of modes at the thread level is that they represent different execution sequences based on different input from ports. If desired, we can use the Behavior Annex of AADL to specify the conditions under which alternative execution sequences are taken, based on the value of the incoming goal. Use of the annex leads to the view that the supervisor provides the desired goal to the thread as a data value, and only the most recent value is relevant. The supervisor thus communicates the desired goal through a data port. If goal delivery occurs over a network whose protocol does not guarantee delivery, it may be desirable to resend such goals periodically to ensure that the control layer responds to the request even under transient delivery fault conditions.

6.2 Goal Network-Based Workload Analysis

During the goal network planning process, the goal network is elaborated into subgoals, and goal subnets are merged. Goals in a goal network have constraints on time and on both desired state values and observed state values (state estimates). Constraints help identify whether goals are reachable and which of multiple alternative tactics to use to reach a goal. As the goal network is elaborated into a scheduled goal network, it is checked for inconsistency. A verification procedure for such goal networks is discussed in Braman [Braman 2007].

The workload of the set of tasks at the control layer is determined by the set of threads that are active and the mode that each thread operates in. In our model, we do not disable and enable individual control layer threads through modes at the enclosing process. Therefore, all control layer threads are active, but may operate at different modes at different times, resulting in different execution time demands.

As part of an instance model, AADL records the set of *System Operation Modes (SOM)* that are feasible in the given system. An SOM is the set of current modes of all modal components in the instance model. In our case, an SOM is the set of current modes of all control layer threads at any given point in time, plus any other threads with modes in the system. Examples of other threads are the threads at the planning layer performing goal planning, goal execution, and goal monitoring (see Section 7.1 for details). In this section, we focus on determining the feasible current mode combinations of the control layer threads based on a scheduled goal network.

Given a set of SOMs in an instance model, the OSATE tool set supports the traversal of the instance model for each of the SOMs to drive an analysis in such a way that the analysis tool can be ignorant of the SOMs. In other words, the instance model is traversed visiting only those component and connection instances that are active according to the selected SOM, and property values that apply to the currently selected SOM are passed to the analysis tool. This action is the case for the resource budget analysis and the scheduling analysis plug-ins available as part of the default configuration of OSATE.

We can determine the service levels (i.e., the modes) of the control layer threads from the goal network in a manner that is similar to the algorithm used in building and verifying a scheduled goal network [Braman 2007, Bennett 2006]. By default, the control layer threads are operating in standby mode. By "walking" the goal network in a simulated execution, we determine all threads that receive a non-standby *xgoal* (i.e., a non-standby mode). First, we determine the initial set of *xgoals* (SOM) that are to be executed concurrently by identifying those goal states in the goal network that do not have any ancestors. Given this starting set, we traverse the goal network. For each successor goal state that is independent of other goal states, we get a new SOM by updating the current mode in the SOM for the thread(s) whose *xgoals* change. As we encounter time-coordinated goal states, we update the current modes of all threads, whose *xgoals* change across the coordinated goal states. As we encounter alternative tactics in the form of alternative branches out of a goal state, we elaborate each alternative separately in determining the next SOM. The list of feasible SOMs is maintained as a unique list (i.e., we need to consider each different SOM only once in our workload analysis).

The result of a workload analysis across all feasible SOMs allows us to determine the maximum workload while the system is still schedulable. We can apply this analysis before and during a

mission. Before a mission, we can use this analysis on a set of planned mission scenarios to determine an appropriately dimensioned computer platform with sufficient margin. We can perform this analysis before committing the goal network to the goal executive. If the goal network is not schedulable in terms of processor utilization we would have to consider replanning. Finally, we can use the result of the analysis to determine the lowest processor speed to meet the timing requirement while reducing power consumption by the computer hardware. Different processor speeds are modeled as different modes of the processor, either as a preconfigured speed determined by the hardware or as a variable set of speeds determined by the analysis. As it is transitioning through the goal network, the goal executive can initiate the appropriate processor mode in the same way it initiates different thread modes—by sending appropriate *xgoals*. We plan to develop a prototype implementation of this algorithm based on the goal network representation and algorithms used in the MDS and have received access to the code from the JPL.

.

7 Goal Failure Management

In this section, we present analyses of an implementation of the MDS architecture for an example rover wheel control system taken from session four of the MDS tutorials developed at JPL [Bennett 2006]. This example rover contains six independently powered wheels as shown in Figure 35.

For these analyses, we model an MDS software implementation that is executing on a single-processor, single-core computer platform. The analyses include AADL models of the scheduling impacts of goal failures on the goal elaboration and controller functions. *Note that values used in the example are illustrative and should not be taken as representative of any existing MDS implementation.*
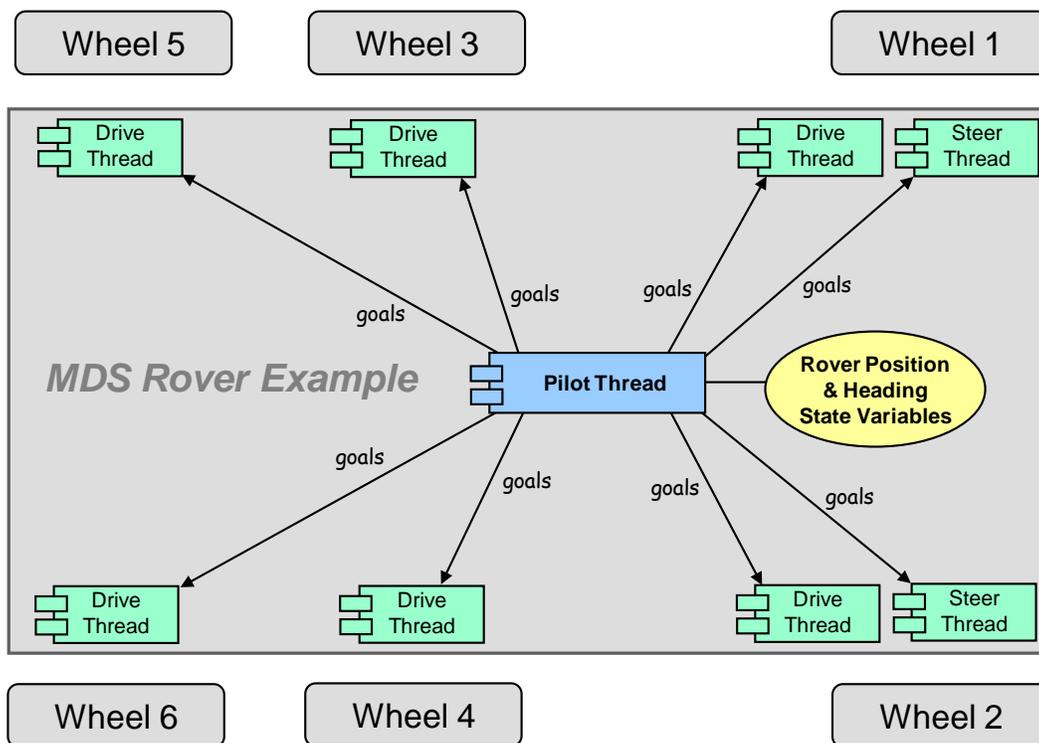
Figure 35: Rover Wheel Example

7.1 Integral Fault Protection with AADL Modes

In this section, we present an analysis of the scheduling and load impact of the MDS integral fault protection. The MDS control system functions are shown in Figure 3 on page 7. Faults are detected within the Goal Execute and Monitor software when a goal or goals are not achieved or when a still-satisfiable goal condition is violated. The response of the system to these situations is the re-elaboration of the system's goal network.

7.1.1 Re-elaboration

In our model, we represent the Goal Executive & Monitor and the Goal Elaboration & Re-elaboration functions as two threads. This is represented in the model as modes of the MDS control system. In the nominal mode, the goal executive and goal monitor are performing their plan execution function by comparing the observations from the state estimation against the conditions in the goal network and by providing setpoints to the state control with given execution times, while the goal planner (shown as Goal Elaboration in the figures below) is relatively idle. We have introduced a goal failure mode and a macro goal failure mode. In each of them, the goal planner has to perform replanning—expending a different amount of effort in each. Similarly, the goal monitor and goal executive expend a little more effort in order to deal with the situation until a revised plan is available.

Table 2 presents the execution times of the monitoring and elaboration threads impacted by goal failure. In this table, we include longer execution times for a macro-goal failure. Note that these values are illustrative and should not be taken as representative of any existing rover implementation.

Table 2:   Illustrative Thread Execution Times for Re-Elaboration

Thread / Thread Group	Mode	Lower Bound (ms)	Upper Bound (ms)
Goal Elaboration	nominal	1	1
Goal Elaboration	goal_failure	3	5
Goal Elaboration	macro_goal_failure	7	9
Goal Executive Monitor	nominal	2	3
Goal Executive Monitor	goal_failure	2	4
Goal Executive Monitor	macro_goal_failure	2	7

Figure 36 presents the AADL text that declares the three modes {*nominal, goal_fail, and macro_goal_fail*} for the control system and the alternative execution times for the Goal Elaboration & Re-elaboration and Goal Executive & Monitor threads. These are contained property associations declarations within the *MDSControlSystem* process. The *MDSControlSystem* includes all of the threads as distinct thread groups for the MDS control. For example, the *elaborator* thread group contains the *Goal_Elaboration_Thread* whose execution time changes with a mode change (i.e., *3 Ms .. 5 Ms* in the *goal_fail* mode).

```
modes
      nominal: initial mode ;
      goal_fail: mode ;
      macro_goal_fail: mode ;
   properties
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 1 Ms ap-
plies to elaborator.Goal_Elaboration_Thread in modes (nominal);

      AADL_Properties::Compute_Execution_Time => 3 Ms .. 5 Ms ap-
plies to elaborator.Goal_Elaboration_Thread in modes (goal_fail);

      AADL_Properties::Compute_Execution_Time => 7 Ms .. 9 Ms ap-
plies to elaborator.Goal_Elaboration_Thread in modes (ma-
cro_goal_fail);
```

```
       AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms ap-
plies to Goal_Executive_Monitor.Goal_Executive_Monitor_Thread in
modes (nominal);

       AADL_Properties::Compute_Execution_Time => 2 Ms .. 4 Ms ap-
plies to Goal_Executive_Monitor.Goal_Executive_Monitor_Thread
in modes (goal_fail);

       AADL_Properties::Compute_Execution_Time => 2 Ms .. 7 Ms ap-
plies to Goal_Executive_Monitor.Goal_Executive_Monitor_Thread
in modes (macro_goal_fail);
```

Figure 36: AADL Text of Mode Configurations

Our analysis assesses the computational burden using the capabilities provided by an OSATE plug-in that can bind threads to processors and determine the schedulability of a configuration. The results of the analysis for each of the modes involved in re-elaboration are presented in Figure 37. The total load includes all of the threads involved in the control system. For this illustrative example, the load increase for a macro goal failure almost consumes the available single processor computational resource.

Processor Capacities	Thread Bindings	Message Bindings	Network Capacities	AADL Prop ◀ ▶
Processor	% Load	% Available/Overload		
Rover_Goal_Failure_...	83%	17%		

Nominal Mode Results

Processor Capacities	Thread Bindings	Message Bindings	Network Capacities	AADL Prop ◀ ▶
Processor	% Load	% Available/Overload		
Rover_Goal_Failure_Model_...	90%	10%		

Goal Failure Mode Results

| Processor Capacities	Thread Bindings	Message Bindings	Network Capacities	AADL Prop ◀ ▶
Processor	% Load	% Available/Overload		
Rover_Goal_Failure_Model...	99%	1%		

Macro Goal Failure Results

Figure 37: Scheduling Analysis Results

Table 3 presents an analysis view report example that might be generated for this analysis effort. This report is an artifact produced using the model-based analysis process described in *A Practice Framework for Model-Based Analysis Using the Architecture Analysis and Design Language (AADL)* [NASA IV&V 2009].

Table 3: Complete Analysis View Report

View Identifier: Rover Schedulability-1	Analyses: Using execution properties, assess the schedulability of the MDS system with the experiments and in the three modes {nominal:1, goal failure:2, macro goal failure:3}.
Process Identifier (optional):	
Scope: The complete rover system **Perspective:** Use system process, thread group, and thread components. Include all relevant scheduling properties. **Constraints:** Include only rover processes. **Specific Guidelines:** Extract useful items from existing models. Use MDS reference architecture for the MDS system components.	

Model File Name (*.aadl or *.aaxl): Rover_Goal_Failure_Model.aadl				
Results			
Analysis ID	**Expected Results**	**Actual Results**	**Assessment and Action(s)**
1	Load <= 100%	Load = 83%	Consistent with required design margin of 10%
2	Load <= 100%	Load = 90%	Just meets required design margin of 10%; a re-evaluation of the system should be considered.
3	Load <= 100%	Load = 99%	Exceeds required design margin of 10%; a re-evaluation of the system should be conducted.

7.1.2 Controller Reconfiguration

In this section, we investigate the impact of a new goal network on the controller functions for the rover example shown in Figure 35. The controller functions affected by the goal network restructuring are those for the wheels. These are modeled as threads: a heading-steering thread, two steering threads, and six drive threads within a controller thread group. For this example, we assume good, fair, and poor health states exist and that the new goal network is elaborated upon a change in the health state. The three health states definitions are those found in Braman [Braman 2007].

In the model, once a new goal network has been elaborated, the elaboration and monitoring functions return to their nominal mode. The controller computational changes in response to the newly elaborated goal network are represented by increases in execution frequency and execution times of the controller threads as health degrades (e.g., representing an assumption that threads must run additional models to compensate for less information or resources). These values are summarized in Table 4.

Table 4: Execution Properties for System Health Modes

System Health (controller mode)	Units: milliseconds		
	Good	**Fair**	**Poor**
Heading Steering Period	100	95	90
Heading Steering Computation Time	2 .. 3	5 .. 8	8 .. 10
Wheel Period	100	95	90
Wheel {One, Two} Steer Computation Time	2 .. 5	2 .. 6	5 .. 7
Wheel {One, Two} Drive Period	1 .. 2	2 .. 3	3 .. 6
Wheel {Three, Four, Five, Six} Drive Computation Time	1 .. 2	2 .. 3	3 .. 5
System Load	**68%**	**84%**	**106%**

To model and analyze the impact of a new goal network, we use AADL modes and mode-dependent property associations for the rover wheel controller thread group. Figure 38 presents an excerpted AADL model of the controller thread group. The excerpted model shows only the subcomponents, health modes, and a few of the mode-dependent property associations for the controller thread group.

The results of the analysis of this illustrative example are shown in the last row of Table 4. Again, these values should not be taken as representative of any existing rover implementation. They are used to illustrate that mode-dependent modeling and early analysis of fault response behavior can identify potential problems with a proposed implementation.

```
  thread group implementation controller.basic
    extends MDS_Software::control.basic
    subcomponents
      Heading_Steering_Thread: thread head_steer_controller.rover;
      Wheel_One_Steer_Thread: thread steer.wheel;
      Wheel_Two_Steer_Thread: thread steer.wheel;
      Wheel_One_Drive_Thread: thread drive.wheel;
      Wheel_Two_Drive_Thread: thread drive.wheel;
      Wheel_Three_Drive_Thread: thread drive.wheel;
      Wheel_Four_Drive_Thread: thread drive.wheel;
      Wheel_Five_Drive_Thread: thread drive.wheel;
      Wheel_Six_Drive_Thread: thread drive.wheel;
      Heading_Sem: data Semaphore.Steering;
      Driving_Sem: data Semaphore.Driving;
....
    modes
      good: initial mode ;
      fair: mode ;
      poor: mode ;
    properties
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms ap-
plies to Heading_Steering_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 8 Ms ap-
plies to Heading_Steering_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 8 Ms .. 10 Ms ap-
plies to Heading_Steering_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 5 Ms ap-
plies to Wheel_One_Steer_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 6 Ms ap-
plies to Wheel_One_Steer_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 7 Ms ap-
plies to Wheel_One_Steer_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 5 Ms ap-
plies to Wheel_Two_Steer_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 6 Ms ap-
plies to Wheel_Two_Steer_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 7 Ms ap-
plies to Wheel_Two_Steer_Thread in modes (poor);
```

Figure 38: Rover Wheel Controller Thread Group

7.1.3   Fault Management and the AADL

Through this example, we demonstrate the use of AADL modes to represent goal network re-elaboration and controller reconfigurations resulting from a re-elaborated network. This approach can be used early in the development effort to assess the load of fault management on computation resources. The results of these analyses can provide a foundation for tradeoff assessments in modifying existing designs and defining alternative implementations. As part of a software assur-

ance practice, these analyses can identify potential problems in an existing design or implementation.

8 Summary

In this case study investigation, we have demonstrated that the AADL can

- effectively model MDS top-level constructs (e.g., hardware adapters, separation of estimation and control, the layering of planning and control)

- effectively represent the MDS reference architecture and support an instantiation of this architecture for an example system

- address key MDS architectural themes (e.g., state-based closed loop control, separation of estimation from control and data management from data transport, ground-to-flight migration)

- provide a foundation for the analysis of critical MDS performance elements and system assurance concerns (e.g., latency, task scheduling, integral fault protection)

In addition, we identified critical areas of the MDS architecture for which AADL models and analyses may provide an effective basis for predicting critical architecture properties and defining adaptation guidelines. Most of these issues have been detailed in other sections of this report. In the remainder of this section, we highlight the application of AADL modeling and analysis to address the issues of handling state variables in the application model, investigating flow latency and latency variation, determining the workload generated by a goal network on the processor resource in the computer platform, and modeling integral fault protection.

8.1 State Variables in the Application Model

The MDS treats state variable and goals data as separate entities in the architecture, as shown in Figure 2 on page 7. This may suggest that the state variable and goals data can be represented as a data component(s) accessed by other components and software tasks.

Within AADL, a flow-oriented model can be developed. In this representation, data flow is represented by data connections and flow specifications. We have included an explicit representation of value histories, reflecting the ownership of measurement and command histories by the hardware adapters of the system under control.

This flow-oriented model separates concerns of information flow from the concerns of data transfer and data control. Furthermore, MDS utilizes data state variables as a key concept to represent meta-data about estimated, intended, and projected state.

8.2 Flow Latency Analysis and Latency Variation

We have demonstrated the AADL capability for modeling end-to-end flows and utilizing this capability to perform flow latency analysis. The MDS architecture includes state effects models that reflect the expected data flow from measurements to estimated states, to derived states, and to controllers determining commands. Sensors and actuators may operate at various rates, resulting in a control system whose control components have different rate requirements.

It has been shown in avionics systems that the use of preemptive fixed priority scheduling, when combined with state communication through shared variables, can result in unexpected latency jitter due to variation in workload that induces instability in the control behavior. This may be the case for Rate Monotonic Analysis (RMA) or other scheduling schemes that do not operate on a static time line and can be particularly problematic when state information is kept in a shared data area.

The AADL flow specification capability and the flow latency analysis framework developed by the SEI provide the opportunity to analytically compare latency improvements. This comparison can be especially valuable when

- migrating planning and control capabilities from the ground to the flight system

- investigating the potential risks due to the impact of rate group optimization of threads

- assessing the impact of distributing the execution of threads across multiple cores of multi-core processor chips

8.3 Goal Networks and Workload Analysis

We have illustrated how AADL can be used to develop task-level architecture models of MDS. AADL modes can be used to capture the dynamics of MDS systems, such as the executive of the mission plan (goal network) driving the workload by setting goals for the control layer of MDS. These goals cause the different estimators and controllers to operate in different modes, generating a different resource demand under each mode. We have shown how to map the execution of MDS goal networks into modal AADL representations and then calculate mode-specific workloads.

8.4 Integral Fault Protection

We have illustrated the potential value of the AADL in representing and analyzing integral fault protection. Specifically, we demonstrated how the goal-oriented task planning and execution approach, used as realization of integral fault protection, can be mapped into AADL capabilities for capturing runtime architecture dynamics. Specifically, we modeled goal planning and monitoring functions in the presence of failures using AADL modal modeling and analysis—assessing performance issues in restructuring and executing goal networks in response to a goal failure.

Appendix AADL Textual Representations

MDS REFERENCE ARCHITECTURE

Figure 39 (below and on the next 9 pages) is a listing of an AADL textual model of the MDS.

```
property set RateGroups is
  RateGroups : type enumeration ( EstimatorRategroup,
        ControllerRateGroup, PlanExecutionRateGroup,
        PlanningRateGroup, HWARateGroup);
  AssignedRateGroup : inherit RateGroups::RateGroups
                    applies to (thread, thread group, process, sys-
tem);
end RateGroups;

package MDSData
public
  -- the port groups will get refined for a MDS instance
  -- they will contain specific data ports
  port group SensorReadingsOut
    features
  end SensorReadingsOut;

  port group SensorReadingsIn
    inverse of SensorReadingsOut
  end SensorReadingsIn;

  port group MeasurementsOut
    features
  end MeasurementsOut;

  port group MeasurementsIn
    inverse of MeasurementsOut
  end MeasurementsIn;

  port group StateEstimatesOut
    features
  end StateEstimatesOut;

  port group StateEstimatesIn
    inverse of StateEstimatesOut
  end StateEstimatesIn;

  port group XgoalsOut
    features
  end XgoalsOut;

  port group XgoalsIn
    inverse of XgoalsOut
  end XgoalsIn;

  port group CommandsOut
    features
  end CommandsOut;
```

```
   port group CommandsIn
     inverse of CommandsOut
   end CommandsIn;

   port group RawCommandsOut
     features
   end RawCommandsOut;

   port group RawCommandsIn
     inverse of RawCommandsOut
   end RawCommandsIn;

   data XGoalsTimeLine
   end XGoalsTimeLine;

   data MissionGoals
   end MissionGoals;
end MDSData;

package ValueHistories
      public
      -- this thread group represents the subsystem managing mea-
surement histories
      -- It contains the measurement history store objects and
threads to log and retrieve
      -- the measurements and their history.
      thread group MeasurementHistories
      features
        MeasurementHistory: port group MeasurementHistory;
        Measurements: port group MDSData::MeasurementsIn;
      end MeasurementHistories;

      thread group implementation MeasurementHistories.template
      -- these are to be declared for each measurement
      -- in subcomponents
      -- tempHistory: thread group HistoryTemplate.basic;
      -- connections
      -- tempvalueconn: data port Measurements.tempval -> tempHis-
tory.datavalue;
      -- temphistoryconn: data port tempHistory.ValueHistoryOut ->
MeasurementHistory.temphistory;
      end MeasurementHistories.template;

      -- this template is used to create instances of history stores
and the access methods
      thread group HistoryTemplate
      features
        datavalue : in data port;
        valuehistory: out data port;
      end HistoryTemplate;

      thread group implementation HistoryTemplate.basic
      subcomponents
       Logger: thread HistoryLogger;
       Retriever: thread HistoryRetriever;
       History: data ValueHistoryStore;
```

```
      connections
       valueconn: data port datavalue -> logger.datavalue;
       historystoreconn1: data access History -> log-
ger.ValueHistory;
       historyconn: data port retriever.ValueHistoryOut ->
                 valuehistory;
       historystoreconn2: data access History ->
                 retriever.ValueHistory;
      end HistoryTemplate.basic;


      thread group CommandHistories
      features
        CommandHistory: port group CommandHistory;
        Commands: port group MDSData::CommandsIn;
      end CommandHistories;

      thread group implementation CommandHistories.template
      -- see above
      end CommandHistories.template;


      thread group EstimationHistories
      features
        EstimateHistory: port group EstimateHistory;
        Estimates: port group MDSData::StateEstimatesIn;
      end EstimationHistories;

      thread group implementation EstimationHistories.template
      -- see above
      end EstimationHistories.template;

      thread HistoryLogger
      features
        ValueHistory: requires data access
             ValueHistories::ValueHistoryStore;
        datavalue: in data port;
    end HistoryLogger;


      thread HistoryRetriever
      features
        ValueHistory: requires data access
             ValueHistories::ValueHistoryStore;
        ValueHistoryOut: out data port;
    end HistoryRetriever;

    data ValueHistoryStore
    end ValueHistoryStore;



  -- the next port groups represent the features that provide access
to histories
  -- through ports
  port group MeasurementHistory
    features
```

```
  end MeasurementHistory;

  port group MeasurementHistoryInv
    inverse of MeasurementHistory
  end MeasurementHistoryInv;

  port group CommandHistory
    features
  end CommandHistory;

  port group CommandHistoryInv
    inverse of CommandHistory
  end CommandHistoryInv;

  port group EstimateHistory
    features
  end EstimateHistory;

  port group EstimateHistoryInv
    inverse of EstimateHistory
  end EstimateHistoryInv;

end ValueHistories;

package SystemUnderControl
public
  system SystemUnderControl
    features
      MeasurementsOut: port group MDSData::MeasurementsOut;
      CommandsIn: port group MDSData::CommandsIn;
      CommandHistoryOut: port group ValueHistories::CommandHistory;
      DeviceBus: requires bus access ExecutionHardware::DeviceBus;
      MeasurementHistoryout: port group
          ValueHistories::MeasurementHistory;
  end SystemUnderControl;

  -- The implementation will be refined for a MDS instance
  -- It will contain the actual sensor and actuator devices
  system implementation SystemUnderControl.basic
    subcomponents
      HardwareAdapters: process
          HardwareAdapters::HardwareAdapters.basic;
      PhysicalSystem: system PhysicalPlant;
    connections
      BusAccessConnection1: bus access DeviceBus ->
          PhysicalSystem.DeviceBus;
      PortGroupConnection1: port group HardwareAdap-
ters.RawCommandsOut
          -> PhysicalSystem.RawCommandsIn;
      PortGroupConnection2: port group
          PhysicalSystem.RawMeasurementsOut ->
          HardwareAdapters.RawReadingsIn;
      PortGroupConnection3: port group CommandsIn ->
          HardwareAdapters.CommandsIn;
      PortGroupConnection4: port group HardwareAdap-
ters.MeasurementsOut
          -> MeasurementsOut;
```

```
        PortGroupConnection5: port group
            HardwareAdapters.CommandHistoryOut -> CommandHistoryOut;
        PortGroupConnection6: port group
            HardwareAdapters.MeasurementHistoryOut ->
            MeasurementHistoryout;
    end SystemUnderControl.basic;

    system PhysicalPlant
      features
        RawCommandsIn: port group MDSData::RawCommandsIn;
        RawMeasurementsOut: port group MDSData::SensorReadingsOut;
        DeviceBus: requires bus access ExecutionHardware::DeviceBus;
    end PhysicalPlant;
end SystemUnderControl;

package ExecutionHardware
public
    system MDSHardware
      features
        DeviceBus: provides bus access DeviceBus;
    end MDSHardware;

    -- The implementation will be refined for a MDS instance
    -- It will contain the actual processor and network configuration
    system implementation MDSHardware.basic
    end MDSHardware.basic;

    bus DeviceBus
    end DeviceBus;
end ExecutionHardware;

package HardwareAdapters
public
    process HardwareAdapters
      features
        RawReadingsIn: port group MDSData::SensorReadingsIn;
        MeasurementsOut: port group MDSData::MeasurementsOut;
        MeasurementHistoryOut: port group
            ValueHistories::MeasurementHistory;
        CommandsIn: port group MDSData::CommandsIn;
        CommandHistoryOut: port group ValueHistories::CommandHistory;
        RawCommandsOut: port group MDSData::RawCommandsOut;
    end HardwareAdapters;

    process implementation HardwareAdapters.basic
      subcomponents
        MeasurementHistory: thread group
            ValueHistories::MeasurementHistories;
        SensorAdapters: thread group SensorAdapters;
        CommandStateHistory: thread group
            ValueHistories::CommandHistories;
        ActuatorAdapters: thread group ActuatorAdapters;
      connections
        MeasurementHistoryConn1: port group
          MeasurementHistory.MeasurementHistory -> MeasurementHisto-
ryOut;
        MeasurementConn1: port group SensorAdapters.MeasurementsOut ->
```

```
        MeasurementHistory.Measurements;
      MeasurementConn2: port group SensorAdapters.MeasurementsOut ->
        MeasurementsOut;
      MeasurementConn3: port group RawReadingsIn ->
        SensorAdapters.RawReadingsIn;
      CommandHistoryConn2: port group
        CommandStateHistory.CommandHistory -> CommandHistoryOut;
      CommandConn1: port group CommandsIn ->
        CommandStateHistory.Commands;
      CommandConn2: port group ActuatorAdapters.RawCommandsOut ->
        RawCommandsOut;
      CommandConn3: port group CommandsIn ->
        ActuatorAdapters.CommandsIn;
  end HardwareAdapters.basic;

  thread group SensorAdapters
    features
      MeasurementsOut: port group MDSData::MeasurementsOut;
      RawReadingsIn: port group MDSData::SensorReadingsIn;
  end SensorAdapters;

  thread group implementation SensorAdapters.basic
  end SensorAdapters.basic;

  thread group ActuatorAdapters
    features
      CommandsIn: port group MDSData::CommandsIn;
      RawCommandsOut: port group MDSData::RawCommandsOut;
  end ActuatorAdapters;

  thread group implementation ActuatorAdapters.basic
  end ActuatorAdapters.basic;
end HardwareAdapters;

package ControlSoftware
public
  -- this type is refined for a MDS instance by refining the
  -- classifiers of the features to be instance specific
  thread group controller
    features
      StateEstimatesIn: port group MDSData::StateEstimatesIn;
      EstimateHistoryIn: port group ValueHisto-
ries::EstimateHistoryInv;
      ControlGoalsIn: port group MDSData::XgoalsIn;
      CommandsOut: port group MDSData::CommandsOut;
  end controller;

  thread group implementation controller.basic
  end controller.basic;

  -- see comments regarding the controller
  thread group estimator
    features
      StateEstimatesOut: port group MDSData::StateEstimatesOut;
      MeasurementsIn: port group MDSData::MeasurementsIn;
      CommandsIn: port group MDSData::CommandsIn;
      EstimateHistoryOut: port group ValueHisto-
```

```
ries::EstimateHistory;
      CommandHistoryIn: port group ValueHisto-
ries::CommandHistoryInv;
      XGoalsIn: port group MDSData::XgoalsIn;
  end estimator;

  -- This implementation is refined for a MDS instance by refining
  -- the classifiers of the subcomponents to be instance specific
  thread group implementation estimator.basic
    subcomponents
      EstimationStateHistory: thread group
         ValueHistories::EstimationHistories;
      ActualEstimator: thread group ActualEstimator;
    connections
      EstimateHistoryConn2: port group
        EstimationStateHistory.EstimateHistory -> EstimateHisto-
ryOut;
      EstimateConn1: port group ActualEstimator.EstimatesOut ->
        EstimationStateHistory.Estimates;
      EstimateConn2: port group ActualEstimator.EstimatesOut ->
        StateEstimatesOut;
      EstimateConn3: port group MeasurementsIn ->
        ActualEstimator.MeasurementsIn;
      EstimateConn4: port group CommandsIn ->
        ActualEstimator.CommandsIn;
      EstimateConn5: port group EstimationStateHisto-
ry.EstimateHistory
        -> ActualEstimator.EstimateHistoryIn;
      EstimateConn6: port group CommandHistoryIn ->
        ActualEstimator.CommandHistoryIn;
  end estimator.basic;

  -- this type is refined for a MDS instance by refining the
  -- classifiers of the features to be instance specific
  thread group ActualEstimator
    features
      EstimatesOut: port group MDSData::StateEstimatesOut;
      MeasurementsIn: port group MDSData::MeasurementsIn;
      EstimateHistoryIn: port group ValueHisto-
ries::EstimateHistoryInv;
      CommandHistoryIn: port group ValueHisto-
ries::CommandHistoryInv;
      CommandsIn: port group MDSData::CommandsIn;
  end ActualEstimator;

  -- This process implementation is refined for a MDS instance by
  -- defining the subcomponents representing the actual estimators
incl.
  -- derived state estimators
  thread group implementation ActualEstimator.basic
  end ActualEstimator.basic;

  thread group GoalPlanner
    features
      GoalNetwork: out data port MDSData::XGoalsTimeLine;
      ReplanRequest: in event data port;
      MissionGoals: in data port MDSData::MissionGoals;
```

```
   end GoalPlanner;

   thread group implementation GoalPlanner.basic
   end GoalPlanner.basic;

   thread group GoalExecutive
     features
       GoalNetwork: in data port MDSData::XGoalsTimeLine;
       XGoalsOut: port group MDSData::XgoalsOut;
   end GoalExecutive;

   thread group implementation GoalExecutive.basic
   end GoalExecutive.basic;

   thread group XGoalMonitor
     features
       XgoalsIn: port group MDSData::XgoalsIn;
       StateEstimateHistoryIn: port group ValueHisto-
ries::EstimateHistoryInv;
       replanrequest: out event data port;
   end XGoalMonitor;

   thread group OperatorConsole
     features
       replanrequest: in event data port;
       EstimateHistoryIn: port group ValueHisto-
ries::EstimateHistoryInv;
       MeasurementHistoryIn: port group
         ValueHistories::MeasurementHistoryInv;
       commandhistoryIn: port group ValueHisto-
ries::CommandHistoryInv;
       missiongoals: out data port MDSData::MissionGoals;
   end OperatorConsole;
end ControlSoftware;

package MDSControlSystem
public
  -- This process type is refined for a MDS instance by refining
  -- the classifiers of the features to be instance specific
  process MDSControlSystem
    features
      MeasurementsIn: port group MDSData::MeasurementsIn;
      CommandsOut: port group MDSData::CommandsOut;
      CommandHistoryIn: port group ValueHisto-
ries::CommandHistoryInv;
      MeasurementHistoryIn: port group
        ValueHistories::MeasurementHistoryInv;
  end MDSControlSystem;

  -- This process implementation is refined for a MDS instance by
refining
  -- the classifiers of the subcomponents to be instance specific
  process implementation MDSControlSystem.basic
    subcomponents
      GoalPlanner: thread group ControlSoftware::GoalPlanner;
      GoalExecutive: thread group ControlSoftware::GoalExecutive;
      GoalMonitor: thread group ControlSoftware::XGoalMonitor;
```

```
        StateEstimation: thread group ControlSoftware::estimator;
        StateControl: thread group ControlSoftware::controller;
        OperatorConsole: thread group ControlSoft-
ware::OperatorConsole;
    connections
        Connection1: port group CommandHistoryIn ->
            StateEstimation.CommandHistoryIn;
        Connection2: port group StateEstimation.EstimateHistoryOut ->
            StateControl.EstimateHistoryIn;
        Connection3: port group MeasurementsIn ->
            StateEstimation.MeasurementsIn;
        Connection4: port group StateEstimation.StateEstimatesOut ->
            StateControl.StateEstimatesIn;
        Connection5: port group GoalExecutive.XGoalsOut ->
            StateEstimation.XGoalsIn;
        Connection6: port group GoalExecutive.XGoalsOut ->
            StateControl.ControlGoalsIn;
        Connection7: port group StateControl.CommandsOut ->
            CommandsOut;
        Connection8: port group StateControl.CommandsOut ->
            StateEstimation.CommandsIn;
        Connection9: data port GoalPlanner.GoalNetwork ->
            GoalExecutive.GoalNetwork;
        Connection10: port group StateEstimation.EstimateHistoryOut ->
            GoalMonitor.StateEstimateHistoryIn;
        Connection11: port group GoalExecutive.XGoalsOut ->
            GoalMonitor.XgoalsIn;
        Connection12: event data port GoalMonitor.replanrequest ->
            GoalPlanner.ReplanRequest;
        Connection13: event data port GoalMonitor.replanrequest ->
            OperatorConsole.replanrequest;
        Connection14: data port OperatorConsole.missiongoals ->
            GoalPlanner.MissionGoals;
        Connection15: port group StateEstimation.EstimateHistoryOut ->
            OperatorConsole.EstimateHistoryIn;
        Connection16: port group CommandHistoryIn ->
            OperatorConsole.commandhistoryIn;
        Connection17: port group MeasurementHistoryIn ->
            OperatorConsole.MeasurementHistoryIn;
    end MDSControlSystem.basic;
end MDSControlSystem;

package CompleteMDSSystem
public
  system CompleteMDSSystem
  end CompleteMDSSystem;

  -- This implementation is refined for a MDS instance by refining
  -- the classifiers of the subcomponents to be instance specific.
  -- In this refinement we can select alternative computer platforms
or bindings
  -- for the same MDS instance
  system implementation CompleteMDSSystem.basic
    subcomponents
      MDSControlSystem: process
          MDSControlSystem::MDSControlSystem.basic;
      MDSSystemUnderControl: system
```

```
              SystemUnderControl::SystemUnderControl.basic;
        MDSComputePlatform: system ExecutionHard-
ware::MDSHardware.basic;
    connections
      SystemtoControllerConn1: port group
          MDSSystemUnderControl.MeasurementsOut ->
          MDSControlSystem.MeasurementsIn;
      SystemtoControllerConn2: port group
          MDSSystemUnderControl.CommandHistoryOut ->
          MDSControlSystem.CommandHistoryIn;
      ControllertoSystemConn: port group
          MDSControlSystem.CommandsOut ->
          MDSSystemUnderControl.CommandsIn;
      BusAccessConnection1: bus access MDSComputePlatform.DeviceBus
->
          MDSSystemUnderControl.DeviceBus;
      PortGroupConnection1: port group
          MDSSystemUnderControl.MeasurementHistoryout ->
          MDSControlSystem.MeasurementHistoryIn;
  end CompleteMDSSystem.basic;
end CompleteMDSSystem;
```

Figure 39: An AADL Textual Representation of the MDS Reference Architecture

EXAMPLE ROVER CONTROLLER THREAD GROUP

```
  thread group implementation controller.basic
    extends MDS_Software::control.basic
    subcomponents
      Heading_Steering_Thread: thread head_steer_controller.rover;
      Wheel_One_Steer_Thread: thread steer.wheel;
      Wheel_Two_Steer_Thread: thread steer.wheel;
      Wheel_One_Drive_Thread: thread drive.wheel;
      Wheel_Two_Drive_Thread: thread drive.wheel;
      Wheel_Three_Drive_Thread: thread drive.wheel;
      Wheel_Four_Drive_Thread: thread drive.wheel;
      Wheel_Five_Drive_Thread: thread drive.wheel;
      Wheel_Six_Drive_Thread: thread drive.wheel;
      Heading_Sem: data Semaphore.Steering;
      Driving_Sem: data Semaphore.Driving;
    connections
    -- all the input connections--
    D01: data port Knowledge_In -> Head-
ing_Steering_Thread.goals_input;
    D02: data port Heading_Steering_Thread.goals_output ->
Wheel_One_Steer_Thread.goals_input;
    D03: data port Heading_Steering_Thread.goals_output ->
Wheel_Two_Steer_Thread.goals_input;
    D04: data port Heading_Steering_Thread.goals_output ->
Wheel_Three_Drive_Thread.goals_input;
    D05: data port Heading_Steering_Thread.goals_output ->
Wheel_Four_Drive_Thread.goals_input;
    D06: data port Heading_Steering_Thread.goals_output ->
Wheel_Five_Drive_Thread.goals_input;
    D07: data port Heading_Steering_Thread.goals_output ->
Wheel_Six_Drive_Thread.goals_input;
    D08: data port Heading_Steering_Thread.goals_output ->
Wheel_One_Drive_Thread.goals_input;
    D09: data port Heading_Steering_Thread.goals_output ->
Wheel_Two_Drive_Thread.goals_input;
    D11: data port Intent_In -> Heading_Steering_Thread.intent_in;
    -- all the output connections--
    D10: data port Heading_Steering_Thread.commands_out -> Com-
mands_Out;
    PG01: port group Wheel_One_Steer_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG02: port group Wheel_Two_Steer_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG03: port group Wheel_One_Drive_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG04: port group Wheel_Two_Drive_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG05: port group Wheel_Three_Drive_Thread.commands_out ->
Heading_Steering_Thread.commands_in;
    PG06: port group Wheel_Four_Drive_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG07: port group Wheel_Five_Drive_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    PG08: port group Wheel_Six_Drive_Thread.commands_out -> Head-
ing_Steering_Thread.commands_in;
    DA1: data access Heading_Sem.Data_Access -> Head-
```

```
ing_Steering_Thread.Steer_Data_Access;
      DA2: data access Heading_Sem.Data_Access ->
Wheel_One_Steer_Thread.Steer_Data_Access;
      DA3: data access Heading_Sem.Data_Access ->
Wheel_Two_Steer_Thread.Steer_Data_Access;
      DA4: data access Driving_Sem.Data_Access -> Head-
ing_Steering_Thread.Drive_Data_Access;
      DA5: data access Driving_Sem.Data_Access ->
Wheel_One_Drive_Thread.Drive_Data_Access;
      DA6: data access Driving_Sem.Data_Access ->
Wheel_Two_Drive_Thread.Drive_Data_Access;
      DA7: data access Driving_Sem.Data_Access ->
Wheel_Three_Drive_Thread.Drive_Data_Access;
      DA8: data access Driving_Sem.Data_Access ->
Wheel_Four_Drive_Thread.Drive_Data_Access;
      DA9: data access Driving_Sem.Data_Access ->
Wheel_Five_Drive_Thread.Drive_Data_Access;
      DA10: data access Driving_Sem.Data_Access ->
Wheel_Six_Drive_Thread.Drive_Data_Access;
    modes
      good: initial mode ;
      fair: mode ;
      poor: mode ;
    properties
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Heading_Steering_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 8 Ms
applies to Heading_Steering_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 8 Ms .. 10 Ms
applies to Heading_Steering_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 5 Ms
applies to Wheel_One_Steer_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 6 Ms
applies to Wheel_One_Steer_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 7 Ms
applies to Wheel_One_Steer_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 5 Ms
applies to Wheel_Two_Steer_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 6 Ms
applies to Wheel_Two_Steer_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 5 Ms .. 7 Ms
applies to Wheel_Two_Steer_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_One_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Wheel_One_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 6 Ms
applies to Wheel_One_Drive_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_Two_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Wheel_Two_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 6 Ms
applies to Wheel_Two_Drive_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_Three_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
```

```
applies to Wheel_Three_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 5 Ms
applies to Wheel_Three_Drive_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_Four_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Wheel_Four_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 5 Ms
applies to Wheel_Four_Drive_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_Five_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Wheel_Five_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 5 Ms
applies to Wheel_Five_Drive_Thread in modes (poor);
      AADL_Properties::Compute_Execution_Time => 1 Ms .. 2 Ms
applies to Wheel_Six_Drive_Thread in modes (good);
      AADL_Properties::Compute_Execution_Time => 2 Ms .. 3 Ms
applies to Wheel_Six_Drive_Thread in modes (fair);
      AADL_Properties::Compute_Execution_Time => 3 Ms .. 5 Ms
applies to Wheel_Six_Drive_Thread in modes (poor);
      AADL_Properties::Period => 75 Ms applies to Head-
ing_Steering_Thread in modes (good);
      AADL_Properties::Period => 75 Ms applies to Head-
ing_Steering_Thread in modes (fair);
      AADL_Properties::Period => 75 Ms applies to Head-
ing_Steering_Thread in modes (poor);
      AADL_Properties::Period => 100 Ms applies to
Wheel_One_Steer_Thread in modes (good);
      AADL_Properties::Period => 95 Ms applies to
Wheel_One_Steer_Thread in modes (fair);
      AADL_Properties::Period => 90 Ms applies to
Wheel_One_Steer_Thread in modes (poor);
      AADL_Properties::Period => 100 Ms applies to
Wheel_Two_Steer_Thread in modes (good);
      AADL_Properties::Period => 95 Ms applies to
Wheel_Two_Steer_Thread in modes (fair);
      AADL_Properties::Period => 90 Ms applies to
Wheel_Two_Steer_Thread in modes (poor);
      AADL_Properties::Period => 100 Ms applies to
Wheel_One_Drive_Thread in modes (good);
      AADL_Properties::Period => 95 Ms applies to
Wheel_One_Drive_Thread in modes (fair);
      AADL_Properties::Period => 90 Ms applies to
Wheel_One_Drive_Thread in modes (poor);
      AADL_Properties::Period => 100 Ms applies to
Wheel_Two_Drive_Thread in modes (good);
      AADL_Properties::Period => 95 Ms applies to
Wheel_Two_Drive_Thread in modes (fair);
      AADL_Properties::Period => 90 Ms applies to
Wheel_Two_Drive_Thread in modes (poor);
      AADL_Properties::Period => 100 Ms applies to
Wheel_Three_Drive_Thread in modes (good);
      AADL_Properties::Period => 95 Ms applies to
Wheel_Three_Drive_Thread in modes (fair);
      AADL_Properties::Period => 90 Ms applies to
Wheel_Three_Drive_Thread in modes (poor);
```

```
        AADL_Properties::Period => 100 Ms applies to
Wheel_Four_Drive_Thread in modes (good);
        AADL_Properties::Period => 95 Ms applies to
Wheel_Four_Drive_Thread in modes (fair);
        AADL_Properties::Period => 90 Ms applies to
Wheel_Four_Drive_Thread in modes (poor);
        AADL_Properties::Period => 100 Ms applies to
Wheel_Five_Drive_Thread in modes (good);
        AADL_Properties::Period => 95 Ms applies to
Wheel_Five_Drive_Thread in modes (fair);
        AADL_Properties::Period => 90 Ms applies to
Wheel_Five_Drive_Thread in modes (poor);
        AADL_Properties::Period => 100 Ms applies to
Wheel_Six_Drive_Thread in modes (good);
        AADL_Properties::Period => 95 Ms applies to
Wheel_Six_Drive_Thread in modes (fair);
        AADL_Properties::Period => 90 Ms applies to
Wheel_Six_Drive_Thread in modes (poor);
  end controller.basic;
```

Figure 40: An AADL Textual Representation of a Rover Controller Thread Group

Glossary of Acronyms

Table 5 summarizes relevant acronyms for the case study and this report.

Table 5:   A Summary of Acronyms

Acronym	Definition
AADL	Architecture Analysis and Design Language
CPU	Central Processing Unit
CSV	Comma-Separated Values
IV&V	Independent Verification and Validation
JPL	Jet Propulsion Laboratory
KW	Kilowatts
KWH	Kiliwatt Hour
MBE	Model-Based Engineering
MDS	Mission Data System
mW	Milliwatts
NASA	National Aeronautics and Space Administration
OSATE	Open Source AADL Tool Environment
RMA	Rate Monotonic Analysis
SAE	Society of Automotive Engineers
SARP	Software Assurance Research Program
SEI	Software Engineering Institute
SLOC	Source Lines of Code
SOM	System Operation Mode
SV	State Variable
UML	Unified Modeling Language
V&V	Verification and Validation
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
W	Watts
XML	Extensible Markup Language

References

URLs are valid as of the publication date of this document.

[Bennett 2006]
Bennett, Matthew, Dvorak, Daniel, Horvath, Greg, Ingham, Michel, Morris, Richard, Rasmussen, Robert, & Wagner, David. *State Analysis for Software Engineers Model-Based Systems and Software Engineering*. Tutorial, May 10−11, 2006, California Institute of Technology. https://pub-lib.jpl.nasa.gov/docushare/dsweb/View/Collection-64 (2008).

[Bennett 2008]
Bennett, Matthew, Dvorak, Daniel, Hutcherson, Joseph, Ingham, Michel, Rasmussen, Robert, & Wagner, David. *An Architectural Pattern for Goal-Based Control.* California Institute of Technology, March 2008. https://pub-lib.jpl.nasa.gov/docushare/dsweb/View/Collection-63

[Braman 2007]
Braman, Julia M. B., Murray, Richard M., & Ingham, Michel D. "Verification Procedure for Generalized Goal-based Control Programs. " *AIAA, Infotech Aerospace 2007 Conference and Exhibit*. Rohnert Park, CA (USA), May 7−10, 2007.
http://pdf.aiaa.org/preview/CDReadyMIA07_1486/PV2007_3010.pdf

[Cervin 2006]
Cervin, A., Årzén, K.-E., & Henriksson, D. "Control Loop Timing Analysis Using TrueTime and Jitterbug," 1194−1199. *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design (CACSD)*. Munich, Germany, October 4−6, 2006.

[Dvorak 2000]
Dvorak, Daniel, Rasmussen, Robert, Reeves, & Sacks, Allan. "Software Architecture Themes in JPL's Mission Data System," 259−268, vol. 7. *Proceedings of 2000 IEEE Aerospace Conference*, Big Sky, MT (USA), March 2000.

[Feiler 2007]
Feiler, P. H.& Hansson, J. *Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)* (CMU/SEI-2007-TN-010). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2007. http://www.sei.cmu.edu/library/abstracts/reports/07tn010.cfm.

[Feiler 2008]
Feiler, Peter H. & Hansson, Jörgen. "Impact of Runtime Architectures on Control System Stability." *Proceedings of 4th International Congress on Embedded Real-Time Systems (ERTS 2008)*. Toulouse, France, January 2008.

[Ingham 2004]
Ingham, Michel D., Rasmussen, Robert D., Bennett, Matthew B., & Moncada, Alex C. "Engineering Complex Embedded Systems with State Analysis and the Mission Data System," AIAA 2004−6518. *AIAA 1st Intelligent Systems Technical Conference*. Chicago, IL (USA), September 20-22, 2004.

[Leveson 2004]

Leveson, Nancy. *The Role of Software in Spacecraft Accidents* AIAA Journal of Spacecraft and Rockets, Vol. 41, No. 4, July 2004. http://sunnyday.mit.edu/papers/jsr.pdf

[NASA 2009]

Dvorak, Daniel L., et al. *NASA Study on Flight Software Complexity*. NASA Office of Chief Engineer Technical Excellence Program, February 2009.
http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf

[NASA IV&V 2009]

NASA. "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." NASA IV&V Facility Research Program Results and SARP Results.
http://sarpresults.ivv.nasa.gov/ViewResearch/21/156.jsp (2009).

[SAE AADL 2004/2009]

Society of Automotive Engineers (SAE). "The Architecture Analysis and Design Language (AADL)." *Society of Automotive Engineers (SAE)* Standard AS-5506 (November 2004) Revised in January 2009 as AS-5506A. http://www.sae.org/technical/standards/AS5506A.

[SAE AADL 2006]

Society of Automotive Engineers (SAE). "SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex." *Society of Automotive Engineers (SAE)* Standard AS- 5506/1 (June 2006). http://www.sae.org/technical/standards/AS5506/1

[SAVI 2009]

Feiler P.H., Hansson J., de Niz D., Wrage L. "System Architecture Virtual Integration: An Industrial Case Study", Software Engineering Institute Technical Report, CMU/SEI-2009-TR-017, Nov 2009. http://www.sei.cmu.edu/library/abstracts/reports/09tr017.cfm

[SEI 2010]

Software Engineering Institute. *Open Source Tools*: *Open Source AADL Tool Environment (OSATE)*. Software Engineering Institute, Carnegie Mellon University.
http://www.sei.cmu.edu/dependability/tools/osate/ (2010). Download is available at http://www.aadl.info/aadl/currentsite/tool/osate-down.html

REPORT DOCUMENTATION PAGE

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** May 2010	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE** Case Study: Model-Based Analysis of the Mission Data System Reference Architecture	5. **FUNDING NUMBERS** FA8721-05-C-0003

6. **AUTHOR(S)** Peter H. Feiler, David Gluch, Kurt Woodham

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2010-TR-003

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2010-003

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS	12B **DISTRIBUTION CODE**

13. ABSTRACT (MAXIMUM 200 WORDS)

This report documents the results of applying the Architecture Analysis and Design Language (AADL) to the Mission Data System (MDS) architecture. The work described in this case study is part of the National Aeronautics and Space Administration (NASA) Software Assurance Research Program (SARP) research project "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." The report includes discussion of modeling and analyzing the MDS reference architecture and its instantiation for specific platforms. In particular, it focuses on modeling aspects of state-based system behavior in MDS for quantitative analysis. Three different types of state-based system models are considered: closed loop control, goal-oriented mission plan execution, and fault tolerance through mission replanning. This report demonstrates modeling and analysis of the MDS reference architecture as well as instantiations of the reference architecture for a specific mission system.

14. **SUBJECT TERMS** Mission data system, AADL, model-based software assurance	15. **NUMBER OF PAGES** 82

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL